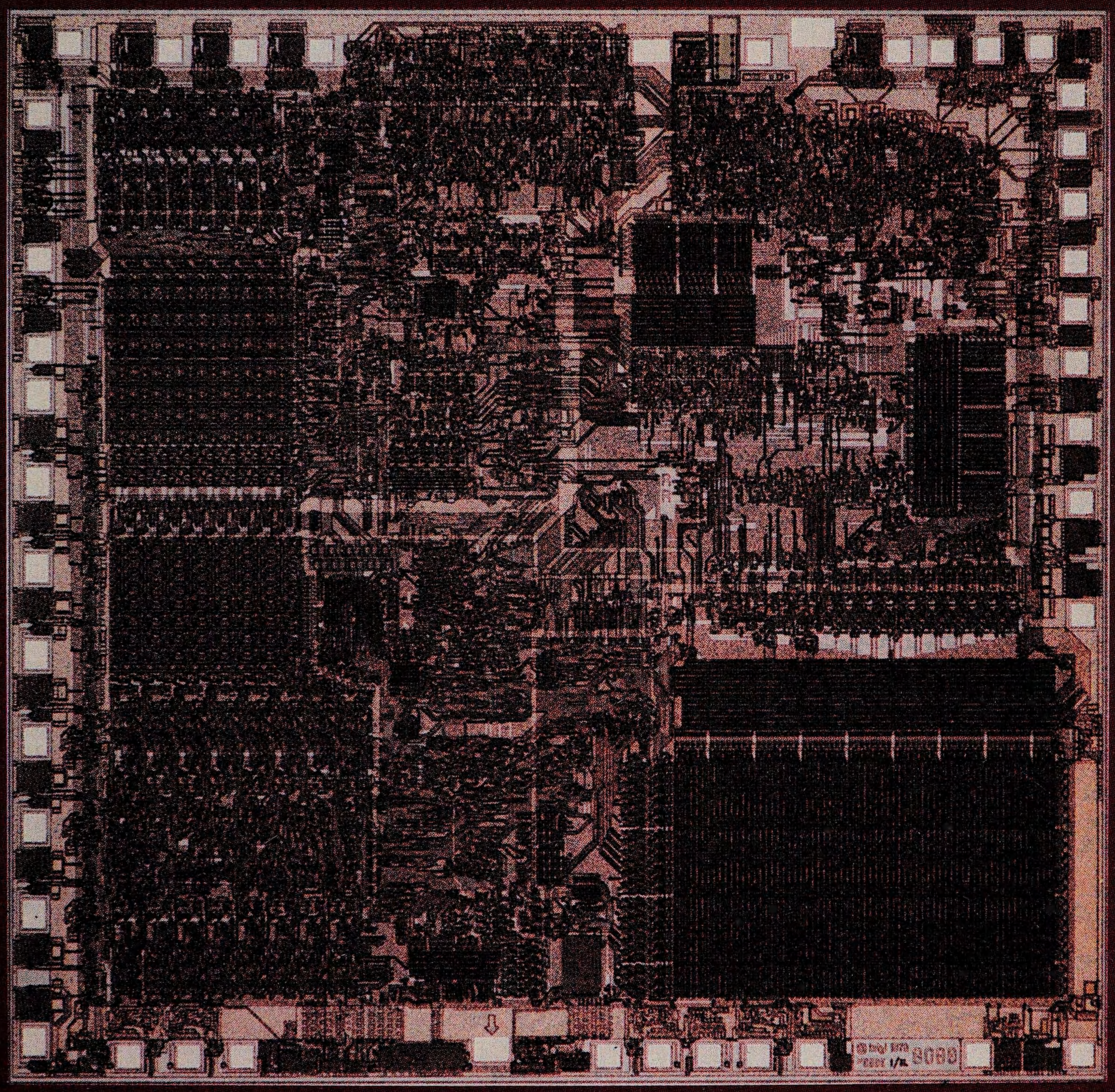
IBM PC & XT
ASSEMBLY LANGUAGE
A GUIDE FOR PROGRAMMERS

Enhanced and Enlarged



Brady

Leo J. Scanlon

IBM PC & XT Assembly Language

A Guide for Programmers

Enhanced and Enlarged

Leo J. Scanlon

Brady Communications Company, Inc.
A Simon&Schuster Publishing Company
New York, NY 10020

IBM PC & XT Assembly Language: A Guide for Programmers, Enhanced and Enlarged

Copyright © 1985 by Brady Communications Company, Inc.
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Brady Communications Company, Inc., Simon&Schuster Building, 1230 Avenue of the Americas, New York, NY 10020.

Library of Congress Cataloging in Publication Data

Scanlon, Leo J.
IBM PC & XT assembly language.

Rev. ed. of; IBM PC assembly language. c1983.
Includes index.
1. IBM Personal Computer—Programming.
2. IBM Personal Computer XT—Programming.
3. Assembler language (Computer program language)
I. Scanlon, Leo J. IBM PC assembly language.
II. Title. III. Title: IBM PC and XT assembly language.
QA76.8.I2594S3 1985 001.64'2 85-11654

ISBN 0-89303-575-0

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95 1 2 3 4 5 6 7 8 9 10

Acquisitions Editor: Terrell Anderson
Production/Design: Jack Straw Enterprises
Art Director/Cover Design: Don Sellers
Assistant Art Director: Bernard Vervin
Manufacturing Director: John A. Komsa

Typesetter: Alexander Typesetting, Inc., Indianapolis, IN
Printer: R. R. Donnelley & Sons Company, Harrisonburg, VA
Typefaces: Palatino (text), Helvetica (display), OCR-B (computer-related material)

Contents

Preface / ix

0 A Crash Course in Computer Numbering Systems / 1
0.1 The Binary Numbering System / **1**
Eight Bits Make a Byte / **3**
Adding Binary Numbers / **4**
Signed Numbers / **4**
0.2 The Hexadecimal Numbering System / **6**
Uses of Hexadecimal Numbers / **7**
Study Exercises / **8**

1 Introduction to Assembly Language Programming / 9
1.1 What is Assembly Language? / **9**
1.2 Evolution of the 8088 / **10**
1.3 Overview of the 8088 Microprocessor / **11**
Addressing / **11**
Software Features / **12**
Input/Output Space / **13**
Memory Allocation / **14**
Interrupts / **14**
Data and Address Buses / **15**
1.4 Internal Registers / **15**
Data Registers / **15**
Segment Registers / **16**
Pointer and Index Registers / **17**
Instruction Pointer / **17**
Flags / **18**
Study Exercises / **20**

2 Using an Assembler / 21
2.1 Introducing the Assembler / **21**
The IBM Macro Assembler / **21**
2.2 Developing an Assembly Language Program / **22**
Editor / **22**
Assembler / **23**
Linker (LINK) / **23**
A Debugging Program (DEBUG) / **24**
"Top-Down" Program Design / **24**
2.3 Source Statements / **25**
Constants in Source Statements / **25**
2.4 Assembly Language Instructions / **26**
Label Field / **26**
Mnemonic Field / **28**
Operand Field / **28**
Comment Field / **28**
2.5 Pseudo Operations / **29**
Data Pseudo-Ops / **30**
Listing Pseudo-Ops / **39**
2.6 Operators / **41**

Arithmetic Operators / **44**
Logical Operators / **44**
Relational Operators / **46**
Value-Returning Operators / **47**
Attribute Operators / **48**
2.7 Entering, Assembling, and Running a Program / **50**
Creating a Working Assembler Disk / **50**
Example Program / **51**
Entering the Program / **53**
Assembling the Program / **55**
Listing the Source Program / **55**
Creating the Run File / **56**
Running the Program / **60**
Advanced Listing Options / **63**
2.8 Models for Constructing Programs / **65**
2.9 Advanced Pseudo-Ops / **67**
Data Pseudo-Ops / **68**
Conditional Pseudo-Ops / **70**
Listing Pseudo-Ops / **73**
2.10 Key Point Summary / **75**
Study Exercises / **78**

3 The 8088 Instruction Set / 79
3.1 About This Chapter / **79**
3.2 Addressing Modes / **79**
Register and Immediate Addressing / **81**
Memory Addressing Modes / **82**
3.3 Instruction Types / **88**
3.4 Data Transfer Instructions / **88**
General-Purpose Instructions / **92**
Input/Output Instructions / **97**
Address Transfer Instructions / **97**
Flag Transfer Instructions / **99**
3.5 Arithmetic Instructions / **100**
Arithmetic Data Formats / **100**
Addition Instructions / **102**
Subtraction Instructions / **106**
Multiplication Instructions / **111**
Division Instructions / **112**
Sign-Extension Instructions / **114**
3.6 Bit Manipulation Instructions / **115**
Logical Instructions / **115**
Shift and Rotate Instructions / **118**
3.7 Control Transfer Instructions / **121**
Unconditional Transfer Instructions / **122**
Conditional Transfer Instructions / **128**
Iteration Control Instructions / **132**
3.8 String Instructions / **134**
Repeat Prefixes / **136**

Move-String Instructions / **137**
Compare-String Instructions / **140**
Scan-String Instructions / **142**
Load-String and Store-String Instructions / **143**
3.9 Interrupt Instructions / **144**
3.10 Processor Control Instructions / **147**
Flag Operations / **148**
External Synchronization Instructions / **148**
No Operation Instruction / **149**
3.11 Key Point Summary / **150**
Study Exercises / **152**

4 High-Precision Mathematics / 155
4.1 Multiplication / **155**
Unsigned 32-Bit × 32-Bit Multiply / **156**
Sign 32-Bit × 32-Bit Multiply / **159**
4.2 Division / **161**
Dealing with Overflow / **162**
4.3 Square Root / **165**
Study Exercise / **167**

5 Operating on Data Structures / 169
5.1 Unordered Lists / **169**
Adding an Element to an Unordered List / **170**
Deleting an Element from an Unordered List / **171**
Maximum and Minimum Values in an Unordered List / **172**
5.2 Sorting Unordered Data / **175**
Bubble Sort / **175**
5.3 Ordered Lists / **181**
Searching an Ordered List / **181**
Adding an Element to an Ordered List / **186**
Deleting an Element from an Ordered List / **188**
5.4 Look-Up Tables / **189**
Look-Up Tables Can Replace Equations / **189**
Look-Up Tables Can Perform Code Conversions / **194**
Jump Tables / **196**
5.5 Text Files / **197**
Study Exercises / **200**

6 Using the System Resources / 201
6.1 System Memory / **201**
6.2 BIOS Interrupts / **202**
8088 Interrupt Vectors / **205**
8259 Interrupt Vectors / **207**
BIOS Entry Points / **208**
User-Supplied Routines / **217**
Data Table Pointers / **218**
6.3 DOS Interrupts / **218**
Type 21 Function Calls / **223**
DOS 2 Error Message Program / **232**

Write-Protecting Files / **234**
6.4 Keyboard Operations / **237**
ASCII / **237**
How the PC Keyboard Works / **238**
Character Codes and Scan Codes / **240**
Keyboard Interrupts / **244**
6.5 ASCII/Binary Code Conversions / **247**
Converting an ASCII String to Binary / **247**
Converting a Binary Number to ASCII / **254**
Study Exercises / **255**

7 Graphics Made Easy / 257
7.1 Display Modes / **257**
Monochrome Adapter / **257**
Color/Graphics Adapter / **257**
7.2 Displaying Characters / **258**
The Character Set / **258**
Display Commands / **260**
A Simple Display Operation / **261**
7.3 The Basics of Animation / **261**
The Old Moving Face Trick / **265**
7.4 Creating Complex Shapes with Shape Tables / **266**
General Display Procedure / **267**
A Mover-and-Shaper / **269**
Study Exercises / **272**

8 Let There Be Sound! / 275
8.1 How the Speaker Works / **275**
8.2 Programming the Speaker / **275**
The BIOS "BEEP" Procedure / **276**
A More General Sound Generator / **276**
8.3 Music, Music, Music / **277**
Music-Playing Procedure / **280**
Music from the Keyboard / **283**

9 Macros / 287
9.1 Introduction to Macros / **287**
Macros vs Procedures / **287**
Macros Speed Up Programming / **288**
Contents of Macros / **289**
9.2 Macro Pseudo-Ops / **290**
General-Purpose Pseudo-Ops / **290**
Repeat Pseudo-Ops / **293**
Conditional Pseudo-Ops / **294**
Listing Pseudo-Ops / **296**
9.3 Macro Operators / **296**
9.4 Defining Macros in Source Programs / **297**
9.5 Macro Library / **299**
Creating a Macro Library / **300**
Guidelines for Defining Macros / **300**

Reading a Macro Library Into a Program / **300**
Macro Descriptions / **301**
Macro Definitions / **309**

10 Object Libraries / 327
10.1 Building an Object Library / **327**
10.2 Operating on Object Libraries / **328**
Getting a Directory of a Library / **329**
10.3 Using Object Libraries / **329**

11 Structured Programming / 331
11.1 Structure Statements and Logic Flow Structures / **332**
Conditions for Logic Flow Structures / **332**
11.2 The IF Structure / **334**
The ELSE Option / **334**
How the IF Structure Operates / **335**
Operand Options / **336**
11.3 The DO Structure / **338**
DO UNTIL / **338**
DO WHILE / **339**
DO COMPLEX / **341**
Operand Options / **341**
11.4 The SEARCH Structure / **343**
SEARCH UNTIL / **343**
SEARCH WHILE / **345**
SEARCH COMPLEX / **346**
Operand Options / **347**
11.5 Restrictions on Using NCXZ and CXZ / **348**
11.6 Preparing Structured Programs / **349**
The Procedure / **349**
Using SALUT / **350**
How SALUT Reformats Programs / **351**

12 8087 Math Coprocessor / 353
12.1 Internal Registers / **354**
The 8087's Stack / **354**
Floating-Point Format / **354**
12.2 Data Types / **355**
12.3 Instruction Set / **356**
12.4 8087 Programming with the Macro Assembler / **358**
Constants / **359**
Data Definition Pseudo-Ops / **359**
12.5 Summary / **360**

Answers to Study Exercises / 361
A Hexadecimal/Decimal Conversion / 371
B IBM PC Display Characters / 372
C 8088 Instruction Times / 373
D 8088 Instruction Set Summary / 380
E Disk User's Guide / 384
Index / 389

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Note to Authors

Have you written a book related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. Brady produces a complete range of books for the personal computer market. We invite you to write to Terrell Anderson, Senior Editor, Simon & Schuster, General Reference Group, 1230 Avenue of the Americas, New York, NY 10020.

Registered Trademarks

- IBM Personal Computer, Personal Editor, PC/XT, and AT are trademarks and IBM is a registered trademark of International Business Machines Corp.
- Apple II is a registered trademark of Apple Computer Inc.
- WordStar is a trademark of MicroPro International Corp.
- Microsoft is a registered trademark of Microsoft Corp.
- MultiMate is a trademark of Multimate International Corp.
- Commodore 64 and VIC-20 are trademarks of Commodore Business Machines.
- TRS-80 is a trademark of Tandy Corporation.

Preface

Why Assembly Language?

Many people write all of their computer programs in one of the so-called "high-level languages," particularly BASIC. BASIC is easy to learn, easy to use, and fast enough for most computing tasks. That being the case, why would anyone want to use any other language? One reason is that BASIC, like human languages, is not well-suited to everything. Some tasks are much easier in other languages. Imagine, for example, trying to describe fine cooking without some French words or symphonies without some Italian terms. Similarly, special computing tasks like graphics, music, or word processing are often easier in special languages.

Furthermore, BASIC is quite slow. The term "slow" may surprise the beginner, since short programs seem to run instantaneously. However, problems occur in the following situations:

1. When large amounts of data are involved. Notice how slow BASIC is when a program must, for example, sort a long list of names and addresses or accounts. Similarly, BASIC is quite slow when a program must search though a 50-page report or keep inventory records on thousands of items.
2. When graphics is involved. If a program is drawing a picture on the screen, it must work quickly or the delay is intolerable. If objects in the picture are supposed to move, the program must be fast enough to make the motion look natural. This is particularly difficult when the picture contains many objects (such as space ships, base stations, and alien invaders), all of which are moving in different directions.
3. When a lot of decisions or "thinking" is required. This is often necessary in complex games like checkers or chess. The program has to try many possibilities and decide on a reasonable move. Obviously, the more possibilities there are and the more analysis required, the longer it will take the computer to move.

Why is BASIC slow? In the first place, the computer actually translates each BASIC statement into simple internal commands (so-called machine or assembly language). It does this every time it runs the BASIC program. Thus, much of the computer's time is spent translating the program, not running it.

There are versions of BASIC called *compilers* that perform the translation once and then save the translated version. However, BASIC will still be slow because of its mechanical nature. It is really like an automobile with an automatic transmission; no amount of coaxing can ever get you the performance or fuel economy that a skillful driver can achieve with a manual transmission. The human being is simply a more flexible, more skillful, and smarter operator than the automatic transmission or the BASIC interpreter or compiler.

Assembly language is the computer equivalent of a manual transmission. It gives the programmer greater control over the computer at the cost of more work, more detail, and less convenience. Like an automatic transmission, BASIC is good enough most of the time for most programmers. But for those who must get maximum performance from their computers, assembly language is essential. You will find that most complex games, graphics programs, and large business programs are written at least partially in assembly language.

Even if assembly language *is* your likely choice, you may be wondering whether you have enough background to learn assembly language programming. You *do* if you have done some programming of *any* kind. If you know BASIC or some other high-level language, that's fine. If you have developed programs in an assembly language, that's even better. For the benefit of former high-level language users, the book has two starting points.

If you have never programmed in assembly language, start with Chapter 0, which gives you a "crash course" in the *binary* and *hexadecimal* numbering systems. Otherwise, if you already know what these terms mean and understand how to use them, proceed directly to Chapter 1.

The Contents of This Book

In Chapter 1 we introduce the 8088 microprocessor—the PC's "brain"—and discuss its role in the system.

Chapter 2 discusses assemblers in general and then describes the IBM *Macro Assembler*. All PC-compatible assemblers do the same job: they translate instructions *you* can understand into numeric codes the computer's microprocessor understands. Therefore, most of the principles we describe will apply to any other assembler you might have. Chapter 2 also presents a simple program, and tells you how to enter it into the computer, assemble it, and execute (run) it.

Chapter 3 describes the 8088's instruction set, the assembly language commands you use to communicate with your PC. This book treats the instructions in functional groups, rather than alphabetically. That is, we group add with subtract, multiply with divide, and so on. Through this approach you not only get to *understand* what the instructions do, but you also appreciate how they fit together.

Chapter 4 tells you how to combine instructions to perform extended math operations that the microprocessor's instruction set doesn't provide directly. Chapter 5 covers operations on lists and tables.

The PC and XT have a built-in operating program called the Basic I /O System or *BIOS* that regulates the equipment in the system. That is, it does everything necessary for the microprocessor to "talk" to the keyboard, screen, disk drive, printer, and other peripherals. Hence, BIOS serves as the computer's "chief administrator." BIOS contains many useful features that

can save you hours of programming time. Chapter 6 tells you how to take advantage of them.

Chapters 7 and 8 discuss programs that display simple graphics shapes on the screen and generate sounds (and, eventually, *music!*) through the PC's built-in speaker.

Chapter 9 covers *macros*. A macro is a miniprogram inserted in a main program simply by mentioning its name. Macros can make assembly language programs nearly as easy to develop as BASIC programs. This chapter also includes a "library" of more than 30 useful macros.

Chapter 10 describes the use of *object libraries*, disk files that contain assembled subroutines. An object library is, in effect, a collection of ready-made tools your program can choose from. The object library capability is a new feature that IBM added to Version 2 of the Macro Assembler.

Chapter 11 describes another new feature of Macro Assembler Version 2: it lets you develop *structured* assembly language programs. By a "structured" program, we mean one that contains high-level statements that are similar to IF-THEN and FOR-NEXT in BASIC. Structuring can both speed up your programming effort and make your programs easier to understand.

Finally, in Chapter 12 we discuss the 8087 Math Coprocessor, an optional chip that can perform complex arithmetic operations.

The book provides five appendices for your convenience. Appendix A has tables that help you convert hexadecimal numbers to decimal, or vice versa. Appendix B shows the characters and shapes you can display on the screen. Appendices C and D summarize the 8088's instruction set in alphabetical order, and show how long it takes to execute each instruction, how many bytes each instruction occupies in memory, and which status flags it affects. Appendix E is a "Disk User's Guide" that tells you how to use the programs that come on the optional disk.

Study Exercises

Most chapters conclude with a set of questions and programming exercises. Some of these test your understanding of the material in the chapter; others are meant to extend your knowledge of the material into additional, related topics.

What You Need with This Book

To use this book, you need an IBM Personal Computer (PC) with at least one disk drive, or an IBM PC XT. You also need two software packages: an *assembler* and IBM's *Disk Operating System (DOS)*.

Supplementary References

This book is designed to complement the Macro Assembler manual and the manuals that come with the computer, so you probably won't need any other reference books. However, every serious PC user should have a copy of the IBM PC (or XT) *Technical Reference* manual, which is an invaluable source for technical details. Among other things, the *Technical Reference* contains a complete and fully-documented listing of the computer's built-in operating program, the BIOS.

If you have DOS 2.1 or later, you might also want to invest in a copy of IBM's *DOS Technical Reference* manual. This has detailed information on disk organization, DOS features you can call from assembly language, and other "goodies."

Finally, for the full details on the chips in the system, you may want to get one or more of the following reference documents:

- *The iAPX 86,88 User's Manual*
- *The 8086 Family User's Manual*
- *iAPX 88 Book*
- *iAPX 88 /10 Data Sheet*

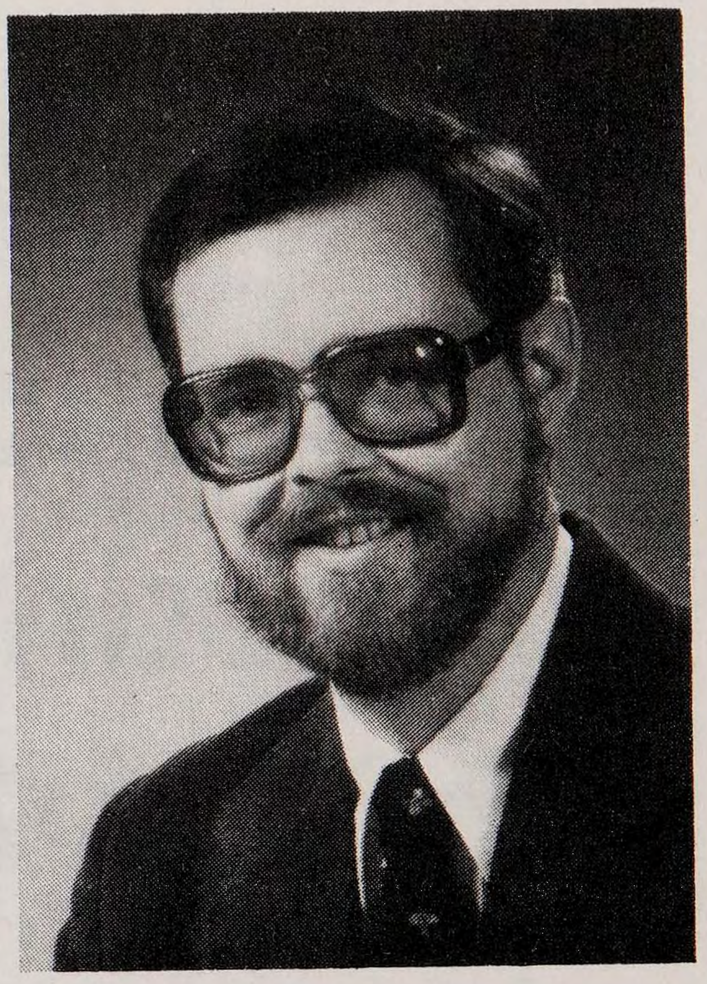
Order them from Intel Corporation, Literature Dept.; 3065 Bowers Ave.; Santa Clara, CA 95051.

Changes in This Edition

Owners of earlier versions of this book will notice quite a few changes in this edition. Some changes reflect new features provided by Version 2.00 of the Macro Assembler; these include new chapters on object libraries and structured programming (Chapters 10 and 11). In addition, I have added a chapter on macros (9) and have enlarged and enhanced many topics based on my own experience and suggestions from readers. In short, I have attempted to make the book more valuable for both learning and reference. Assembly language is a fascinating and efficient programming tool, and I hope you have as much satisfaction using it as I had writing this book.

Leo J. Scanlon
June 1985

About the Author



Leo Scanlon grew up in Pittsburgh, Pennsylvania, and received his BS degree in Aeronautical Engineering from St. Louis University. He also studied Electrical Engineering and Computer Science at the University of Berkeley.

Leo's experience includes technical writing in the minicomputer and microcomputer industries. He also served as technical publications manager with Computer Automation, Inc. in Irvine, CA and Rockwell International Corp. in Anaheim, CA.

Leo is the author of 13 microcomputer books, including *IBM PC & XT Assembly Language, 8086/8088 Assembly Language Programming,* and *Multimate on the IBM PC,* all published by Brady Communications Company.

A freelance writer in Inverness, Florida, he enjoys listening to jazz and boating with his wife, Pat, and sons Roger and Ryan.

This book is dedicated to
my sons, Roger and Ryan, whose
generation will perform "miracles"
in the 21st century.

0

A Crash Course in Computer Numbering Systems

Unless you are visiting from another planet, you have spent your entire life counting things using decimal numbers. Decimal is the *base 10* numbering system, which means it has 10 digits, 0 through 9.

Human beings are quite comfortable counting in decimal (probably because we have 10 fingers and 10 toes), but computers are not. Instead, they count with the base 2 (or *binary*) numbering system, which has only two digits, 0 and 1. Hence, to communicate with the computer at its own level (as you do when you program in assembly language), you must be familiar with binary numbering. Besides binary, assembly language programmers also use another numbering system—base 16 (or *hexadecimal*)—so you must be familiar with it as well.

This chapter is a "crash course" in computer numbering systems for readers who have no previous exposure to them. That's why we call it Chapter *0*. If you already understand binary and hexadecimal numbering, feel free to skip this chapter and begin at Chapter 1.

0.1 The Binary Numbering System

A computer gets all program instructions and data from its *memory*. Memory is comprised of integrated circuits (or "chips") that contain thousands of electrical components. Like light switches, these components have only two possible settings: "On" or "Off." Still, with only these two settings, combinations of memory components can represent numbers of any size. How? Read on.

The On and Off settings of a memory component correspond to the two digits of the *binary numbering system*, the fundamental system for computers. Having only two digits, 1 (On) and 0 (Off), the binary numbering system is a *base 2* system. Again, it differs from the standard decimal numbering system, which has 10 digits (0 through 9).

The switch-like components of memory are called "bits," short for *bi*nary digi*ts*. By convention, a bit that is On has the value 1 and a bit that is Off has the value 0. This appears to be woefully limiting, until you consider that a decimal digit (no, it's not called a "det") can range only from 0 to 9. Just as you can combine decimal digits to form numbers larger than 9, you can combine binary digits to form numbers larger than 1.

As you know, to represent a decimal number larger than 9 requires an additional "tens position" digit. Likewise, to represent a decimal number larger than 99 requires a "hundreds position" digit, and so on. Each decimal digit you add has a *weight* of 10 times the digit to its immediate right.

For example, you can represent the decimal number 324 as

$$(3 \times 100) + (2 \times 10) + (4 \times 1)$$

or as

$$(3 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

In more mathematical terms, each decimal digit is a power of 10 greater than the preceding digit.

A similar rule applies to the binary numbering system. Here, *each binary digit is a power of two greater than the preceding digit.* The rightmost bit has a weight of 2^0 (decimal 1), the next bit has a weight of 2^1 (decimal 2), and so on. For example, the binary number 101 has a decimal value of 5, because

$$\begin{aligned} 101_2 &= (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= (1 \times 4) + (0 \times 2) + (1 \times 1) \\ &= 5_{10} \end{aligned}$$

Do you now understand how binary numbers are constructed? To find the value of any given bit position, you double the weight of the preceding bit position. Thus, the binary weights of the first eight bits are 1, 2, 4, 8, 16, 32, 64, and 128. Figure 0-1 summarizes these weights.

To convert a decimal value to binary, you make a series of simple subtractions. Each subtraction produces the value of a single binary digit (bit).

To begin, subtract the largest possible binary weight from the decimal value and enter a 1 in that bit position. Then subtract the next largest possible binary weight from the result and enter a 1 in *that* bit position. Continue until the result is zero. Enter a 0 in any bit position whose weight cannot be

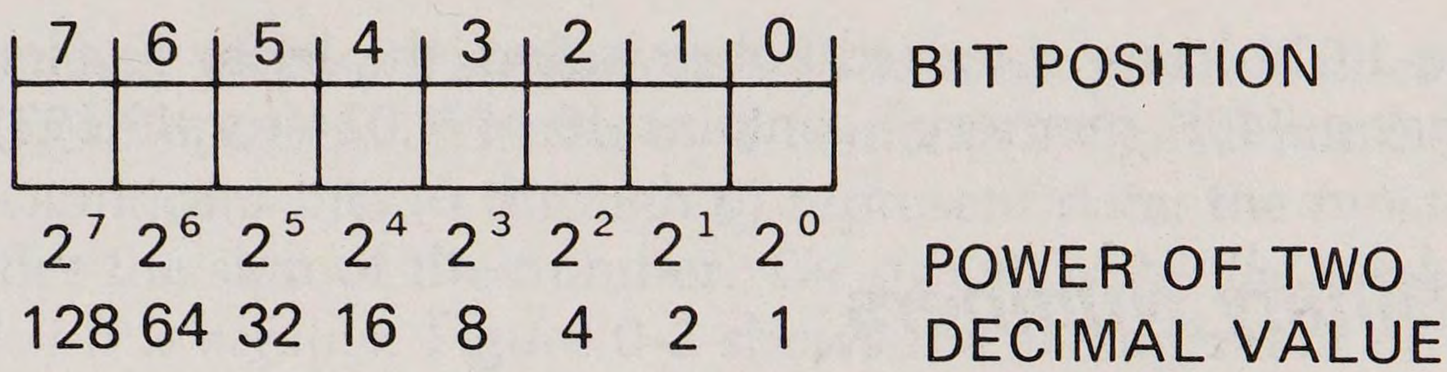


Figure 0-1. Weights of eight binary digits.

subtracted from the current decimal value. For example, to convert decimal 50 to binary:

```
  50
 -32    Bit position 5 = 1
  18
 -16    Bit position 4 = 1
   2
 -  2   Bit position 1 = 1
   0
```

Entering a 0 in the other bit positions (Bits 3, 2, and 0) yields a final result of 110010.

To verify that the binary equivalent of decimal 50 is indeed 110010, add the decimal weights of the "1" positions:

```
  32    (Bit 5)
  16    (Bit 4)
+  2    (Bit 1)
  50
```

Eight Bits Make a Byte

The Apple II, Commodore 64, VIC-20, Radio Shack TRS-80, and many other popular microcomputers are designed around 8-bit *microprocessors*. 8-bit microprocessors are so named because they process information eight bits at a time. To process more than eight bits, they must perform additional operations.

In computer terminology, an 8-bit unit of information is called a *byte*. With eight bits, a byte can represent decimal values from 0 (binary 00000000) to 255 (binary 11111111).

Because a byte is the fundamental unit of processing, microcomputers are described in terms of the number of bytes (rather than bits) their memories can hold. Microcomputer manufacturers construct memory in blocks of 1,024 bytes. This particular quantity reflects the binary orientation of computers in that it represents 2^{10} bytes.

The value 1,024 has a standard abbreviation: the letter *K*. Hence, a computer that has a "48K memory" contains 48 × 1,024 (or 49,152) bytes.

Adding Binary Numbers

You can add binary numbers the same way you add decimal numbers: by carrying any excess from one column to the next. For example, to add the decimal values 7 and 9, you must carry a 1 into the "tens" column to produce the correct result (16). Similarly, to add the *binary* values 1 and 1, you must carry a 1 into the "twos" column to produce the correct result (10).

Adding multi-bit numbers gets slightly more complicated, because you must include a carry from a previous column. For example, this operation involves two carries:

```
  1011
+   11
------
  1110
```

The addition of the rightmost column (1 + 1) produces a result of 0 and a carry of 1 into the second column. With the carry, the addition of the second column (1 + 1 + 1) produces a result of 1 and a carry into the third column.

The general rules for binary addition are shown in this table:

Inputs			*Results*	
Operand #1	***Operand #2***	***Carry***	***Sum***	***Carry***
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Signed Numbers

Until now, we have been discussing how to represent *unsigned numbers* in binary. As we mentioned earlier, each bit in an unsigned number has a weight that reflects its position. The rightmost (or least-significant) bit has a weight of 1, while each more-significant bit has a weight twice that of its predecessor. Therefore, if all eight bits in a byte are 0, the byte has the value 0; if they are all 1, the byte has the value 255.

However, many of your calculations will involve positive or negative values; that is, *signed numbers.* When a byte contains a signed number, only the seven least-significant bits (0 through 6) represent data; the most-significant bit (7) specifies the sign of the number. *The sign bit is 0 if the number is positive or zero and 1 if it is negative.* Figure 0-2 shows the arrangement of signed and unsigned bytes.

When holding a signed number, a single byte can represent positive values between 0 (binary 00000000) and +127 (binary 01111111) and negative values between −1 (binary 11111111) and −128 (binary 10000000).

Two's-Complement Representation

Why is −1 represented in binary as 11111111, instead of as 10000000? The answer is that negative signed numbers are represented in their *two's-complement form.* Computer scientists invented the two's-complement notation to eliminate the problems associated with letting zero have two different values, all zeros ("positive zero") and all zeros with a 1 in the sign position ("negative zero").

To find the binary representation of a negative number (that is, to find its two's-complement form), simply take the positive form of the number and reverse each bit—change each 1 to a 0 and each 0 to a 1—then add 1 to the

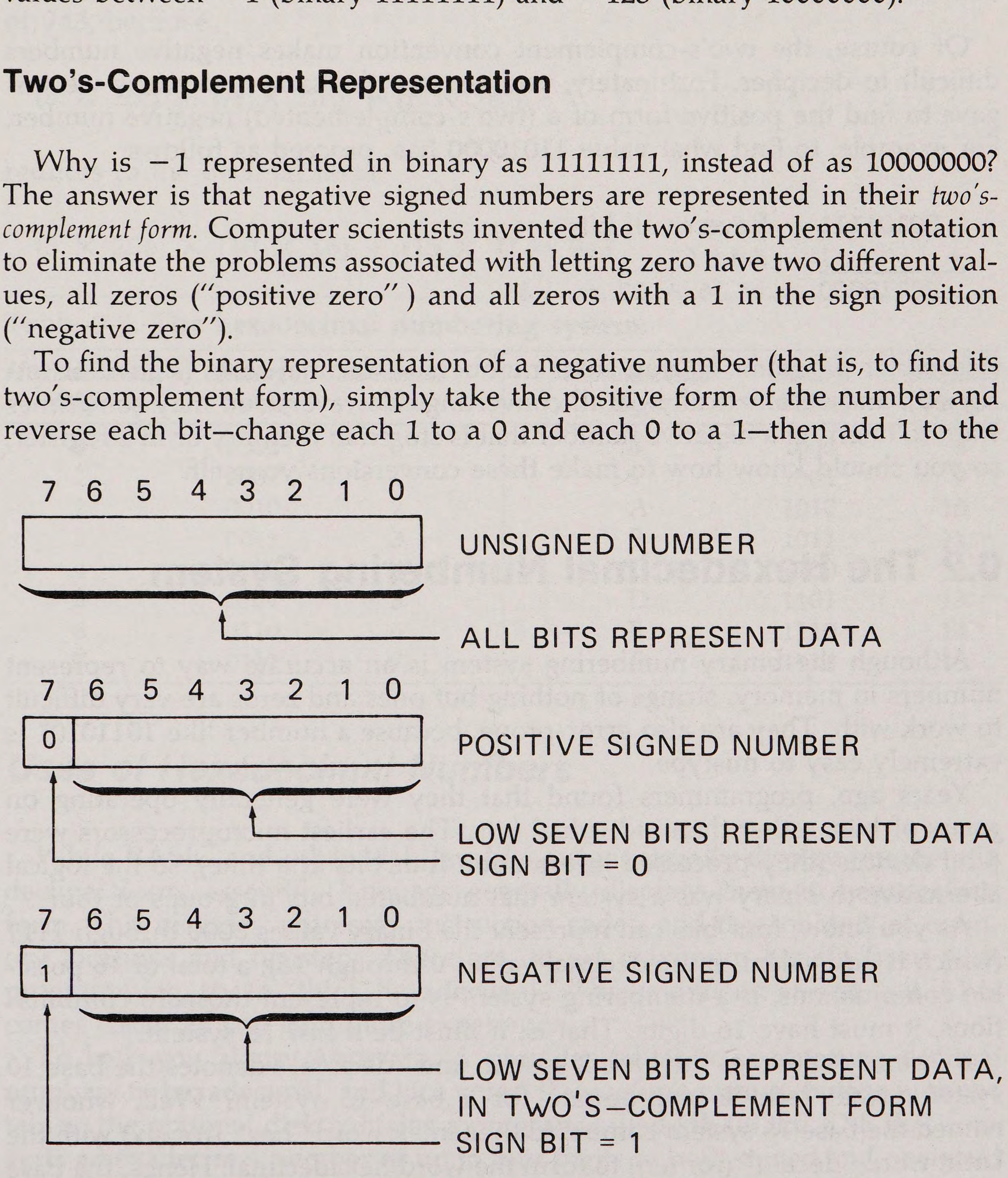


Figure 0-2. Representation of signed and unsigned numbers.

result. The following example shows how to calculate the two's-complement binary representation of −32:

```
  00100000    +32

  11011111    Reverse all bits
+        1    Add 1
--------
  11100000    Two's-complement
```

Of course, the two's-complement convention makes negative numbers difficult to decipher. Fortunately, you can use the same procedure we just gave to find the positive form of a (two's-complemented) negative number. For example, to find what value 11010000 has, proceed as follows:

```
  00101111    Reverse all bits
+        1    Add 1
--------
  00110000    = 16 + 32 = +48
```

Assembler programs let you enter numbers in decimal form (signed or unsigned), and automatically do all converting. However, you may sometimes want to interpret a negative number that is stored in memory or in a register, so you should know how to make these conversions yourself.

0.2 The Hexadecimal Numbering System

Although the binary numbering system is an accurate way to represent numbers in memory, strings of nothing but ones and zeros are very difficult to work with. They are also error-prone, because a number like 10110101 is extremely easy to mistype.

Years ago, programmers found that they were generally operating on *groups* of bits, rather than individual bits. The earliest microprocessors were 4-bit devices (they processed information four bits at a time), so the logical alternative to binary was a system that numbered bits in groups of four.

As you know, four bits can represent the binary values 0000 through 1111 (which is equivalent to the decimal values 0 through 15), a total of 16 possible combinations. If a numbering system is to represent those 16 combinations, it must have 16 digits. That is, it must be a *base 16* system.

If "binary" denotes the base 2 system and "decimal" denotes the base 10 system, what term is appropriate for a base 16 system? Well, whoever named the base 16 system combined the Greek word "hex" (for six) with the Latin word "decem" (for ten) to form the word hexadecimal. Hence, the base 16 system is called the *hexadecimal numbering system.*

Of the 16 digits in the hexadecimal numbering system, the first ten are labeled 0 through 9 (decimal values 0 through 9) and the last six are labeled A through F (decimal values 10 through 15). Table 0-1 lists the binary and decimal values of each hexadecimal digit.

Like binary and decimal digits, each hexadecimal digit has a weight that is some multiple of its base. Since the hexadecimal numbering system is based on 16, each digit has a weight 16 times greater than the digit to its immediate right. That is, the rightmost digit has a weight of 16^0, the next has a weight of 16^1, and so on. For example, the hexadecimal value 3AF has a decimal value of 943, because

$$(3 \times 16^2) + (A \times 16^1) + (F \times 16^0)$$

reduces to the decimal form

$$(3 \times 256) \times (10 \times 16) + (15 \times 1) = 943$$

Table 0-1. The hexadecimal numbering system.

Hexadecimal Digit	*Binary Value*	*Decimal Value*	*Hexadecimal Digit*	*Binary Value*	*Decimal Value*
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

Uses of Hexadecimal Numbers

While BASIC and other high-level languages usually display numbers in decimal form, assembly language generally displays them in hexadecimal form. This includes addresses, instruction codes, and the contents of memory locations and registers. Therefore, to get maximum benefit from your programming, try to "think hexadecimal." This is difficult at first, but it becomes easier as you gain more experience.

To help you along, Appendix A provides tables for converting decimal numbers to hexadecimal, and vice versa. If you don't want to bother with the tables, the optional disk contains a program called HEX2DEC.EXE that converts a hexadecimal number of up to four digits to both signed and unsigned decimal form. To run it, insert the disk with Side 2 (Object Code) up, then

type **hex2dec** and press Return. To leave this program, press Return in response to the prompt.

Study Exercises (answers on page 361)

1. Convert the following decimal values to binary:

   (a) 12 (b) 17 (c) 45 (d) 72

2. Convert the following unsigned binary values to decimal:

   (a) 1000 (b) 10101 (c) 11111

3. How would you represent the three binary numbers in Exercise 2 in hexadecimal?
4. List the decimal equivalent of hexadecimal D8 if:

   (a) D8 represents an unsigned number
   (b) D8 represents a signed number

1

Introduction to Assembly Language Programming

1.1 What is Assembly Language?

Like BASIC, assembly language is a set of words that tell the computer what to do. The words in the assembly language instruction set, however, refer to computer components directly. It's like the difference between telling someone to walk down to the corner and telling them precisely how to move their muscles and maneuver past obstacles. Obviously, a simple command is sufficient most of the time; only athletes or mountain climbers need the more detailed instructions.

Assembly language programs give the computer detailed commands, such as "load 32 into the AX register," "transfer the contents of the CL register into the DL register," and "store the number in the DL register into memory location 3,456." As you see, BASIC and assembly language differ in how you instruct the computer. *With BASIC, you speak in generalities; with assembly language, you speak in specifics.*

Although assembly language programs take more time and effort to write than BASIC programs, they also run much faster. The level of detail is the key here. The idea is the same as an athlete who runs faster or jumps farther by watching every step of what he or she does. Precise form is essential to achieving maximum performance.

Because assembly language requires you to operate on the computer's internal components, you must understand the features and capabilities of the integrated circuit (or "chip") that holds these components, the computer's microprocessor. In this book we will study the microprocessor that is inside the IBM PC and PC/XT: the Intel 8088. Before we get into the details of this chip, let's take a brief look at how it came into being.

1.2 Evolution of the 8088

The earliest microprocessors were 4-bit devices. This means they could transfer only four bits of information at a time. To transfer more than four bits, they had to make several separate transfer operations. Of course, this also made them slow.

The Intel 8008, introduced in 1972, was the first commercial *8-bit* microprocessor (it transferred information eight bits at a time), and is still considered the foremost "first-generation" 8-bit microprocessor. Designed with a calculator-like architecture, the 8008 had an accumulator, six scratchpad registers, a stack pointer (a special address register for temporary storage), eight address registers, and special instructions to perform input and output. In 1973, Intel introduced a "second-generation" version of the 8008, named the 8080.

The 8080 is an upgraded 8008; it has more addressing and input/output (I/O) capability, more instructions, and executes instructions faster. The internal organization is better, too, although Intel maintained the overall 8008 architectural philosophy in the 8080. The 8080 is historically the de facto standard in second-generation microprocessors; the one many people still think of first when someone mentions microprocessors.

By 1976, advances in technology allowed Intel to produce an enhanced version of the 8080, called the 8085. Essentially a repackaged 8080, the 8085 added such features as power-on reset (to initialize the microprocessor), vectored interrupts (to service the needs of peripherals), a serial I/O port (to attach printers and other peripherals), and a single, +5-volt power supply (the 8080 requires two supplies).

By the time the 8085 was introduced, Intel had heavy competition in the 8-bit microprocessor marketplace. Zilog Corporation's 8080 enhancement, the Z80, was catching on, as were non-8080 designs such as the Motorola 6800 and the MOS Technology (now Commodore) 6502. Rather than continue the struggle on the now-diluted 8-bit front, Intel made a quantum leap forward in 1978 by introducing the 8086, a *16-bit* microprocessor that can process data 10 times faster than the 8080.

The 8086 is software-compatible with the 8080 at the assembly language level. This means that with some minimal translation, existing 8080 programs can be reassembled and executed on the 8086. To allow for this, the 8080 registers and instruction set appear as subsets of the 8086 registers and instructions. With this compatibility, Intel capitalized on its experience with the 8080 to gain acceptance in more sophisticated applications.

In the same vein, realizing that many designers still wanted to use the cheaper 8-bit support and peripheral chips in their 16-bit systems, Intel produced a version of the 8086 with the same 16-bit internal data paths, but with an *8-bit data bus* coming out of the chip. This microprocessor, the 8088, is identical to the 8086, except that it takes longer to make 16-bit transfers

because it must perform them with two separate 8-bit transfers. Even so, for applications that deal primarily with 8-bit quantities, the 8088 can approach to within 10 percent of the 8086's processing power!

Thus, if you wish, you can think of your IBM as having a 16-bit microprocessor. (This also means you can refer to the large amount of literature written about the 8086.) With this introduction aside, we can now look at the features of the 8088.

1.3 Overview of the 8088 Microprocessor

Internally, the 8088 holds information in a group of 16-bit boxes called "registers." It has 14 registers in all: 12 data and address registers, plus an instruction pointer (the program-addressing register), and a status (or "flags") register. We can divide the 12 data and address registers into three groups of four registers, called *data registers, pointer and index registers,* and *segment registers.*

Addressing

Since the 8088's instruction pointer and address registers are 16 bits wide, you would expect it to address no more than 64K bytes (65,536 bytes) of memory—the standard range of 8-bit microprocessors. However, the 8088 always produces a 20-bit address. It does this by adding a 16-bit offset to the contents of a segment register multiplied by 16. That is:

$$\text{Physical Address} = \text{Offset} + (16 \times \text{Segment Register})$$

In reality, the 8088 doesn't actually *multiply* the segment register contents by 16, but instead uses them as if they had four extra zero bits (see Figure 1-1). Adding zeros to the end is the same as multiplying, however, because each time you displace a binary number one bit position to the left, you double its value. Thus, displacing the segment register value four bit positions to the left "multiplies" it by 16, since $2 \times 2 \times 2 \times 2 = 16$.

For example, if the offset has the value 10H—where the H suffix means hexadecimal—and the segment register contains 2000H, the 8088 calculates the physical address as follows (operands are shown in binary form):

```
      0000 0000 0001 0000    Offset
+0010 0000 0000 0000 0000    Segment number
-------------------------
  0010 0000 0000 0001 0000   Physical address
```

Therefore, the memory location referenced here has the 20-bit address 20010H.

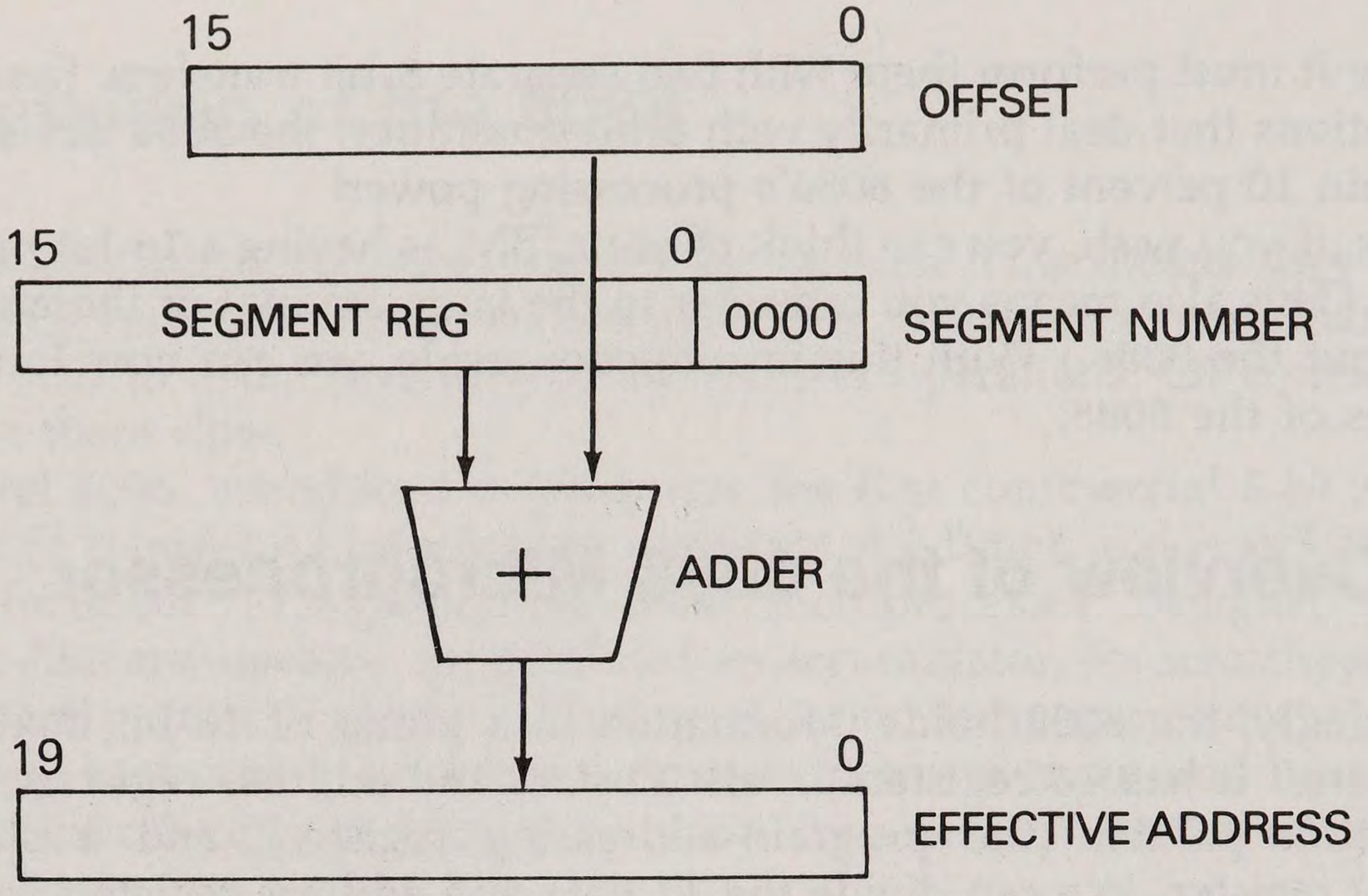


Figure 1-1. How a 20-bit address is generated.

With a 20-bit address at its disposal, the 8088 can access any of 1,048,576 bytes. (We refer to this value as *one megabyte,* but you may think of it as 1024K bytes, if you like.) That's 16 times the addressing range of the 8080!

Segments and Offsets

With most microprocessors, you refer to a memory location by supplying a single number. With the 8088, however, each memory reference requires two terms: a *segment number* and an *offset.* The reason for this odd addressing scheme is that the 8088 requires program instructions and data to be in separate parts of memory; that is, in different segments. Therefore, if you want to address a data location, for example, the 8088 needs the address of the beginning of the data segment (from the data segment register) and the position of the desired location within that segment (its offset). Think of how you find someone's home in a city: you drive to the street where they live (consider it their segment number), then proceed to their address (offset) along that street.

Fortunately, you generally provide only the offset and let the 8088 supply the segment number. We discuss this point in more detail later in the book.

Software Features

The 8088's software features are impressive by any standard, but they will be especially welcomed by programmers who struggled with the earlier 8-bit microprocessors. The 8088 can perform arithmetic operations on signed or

unsigned binary numbers of either eight bits or 16 bits, and on decimal numbers stored in either "packed" (two digits per byte) or "unpacked" (one digit per byte) form. It can also operate on character strings (such as messages) up to 64K bytes long.

The 8088 instruction set has 92 basic instruction types and provides seven different addressing techniques or *modes* for accessing data. The combination of the instruction types, the addressing modes (each with a variety of operand combinations), and the various data types we just mentioned give the 8088 *thousands* of possible instructions to execute. In fact, the combined power of these features allows the 8088 to deliver twice the performance of an 8085, if both processors operate at the same speed.

Measuring Speed

Like electronic watches, microprocessors are powered by quartz crystals. The crystal sends out pulses at a steady, fixed rate, which determines how fast the microprocessor operates. In the PC and PC/XT, the crystal emits 4,770,000 pulses per second.

Computerists don't refer to pulses per second, however, but to *cycles per second,* or *Hertz.* Pulses per second, cycles per second, and Hertz all mean the same thing, but in this book we will use the term Hertz (or, more often, the abbreviation Hz). Hence, we say that the PC has a 4,770,000-Hertz or *4.77-MHz* clock.

At 4.77 MHz (4,770,000 cycles per second), the PC's clock "ticks" every 210 nanoseconds, where a nanosecond (abbreviated ns) is one billionth of a second, or 10^{-9} seconds. We refer to the 210-ns value as the computer's *clock cycle.*

The fastest instructions—for example, one that copies the contents of one register into another—execute in two cycles, or 420 ns. The slowest instruction, a signed 16-bit by 16-bit division, can take up to 206 clock cycles, or about 43 microseconds. (A microsecond, abbreviated *μs,* is one millionth of a second, or 10^{-6} seconds, or 1000 nanoseconds.) As you can see, even this "slowest" instruction executes in the remarkable time of 0.000043 second!

Input/Output Space

In addition to its one-megabyte memory space, the 8088 can address external devices through 65,536 (64K) I/O "ports." It has special input and output instructions that let you communicate with the first 256 ports (0 through 255) directly. Other instructions let you access a port *indirectly,* by putting its identifying number (0 through 65535) in a data register. Like locations in memory, any port can be eight or 16 bits wide.

Memory Allocation

Most of the one-megabyte address space is available for system and user programs, but the 8088 uses some of the highest and lowest locations for special purposes. The highest 16 bytes of memory hold system reset instructions that the 8088 executes when you switch the power on. The lowest 1024 hold the addresses of programs the 8088 executes when an external device interrupts it.

Interrupts

We are all confronted with interruptions from time to time. Some are pleasant, some are unpleasant, and some are neutral. You can ignore some interruptions, if you like; a telephone or doorbell ringing, perhaps, or a child tugging at your sleeve. (On second thought, it's practically impossible to ignore the child!) You simply can't ignore other kinds of interruptions, such as getting a flat tire on an expressway; you must deal with them as soon as possible.

Whatever their cause, interruptions are essentially requests for our attention. In the same way, peripherals in a computer system can request the attention of the processor. The event that makes a processor stop executing its program to perform some requested activity is called an *interrupt.*

Interrupts increase the overall efficiency of a computer system, because they let external devices request the attention of the processor only when necessary. If a system had no interrupts, the processor would have to check every device in the system periodically, to see if any of them needed its attention. This would be like having a telephone with no bell. You would have to pick up the receiver every so often just to find out whether anyone was on the line!

Types of Interrupts

The 8088 can process two types of interrupts: those it can ignore and those it must service as soon as they occur. Interrupts can be generated by external devices, such as disks and other high-speed peripherals; internally, by interrupt-generating instructions; or, under certain conditions, by the 8088 itself.

The 8088 can recognize 256 different interrupts, each with a unique *type code* that identifies it to the microprocessor. The 8088 uses this code (a number between 0 and 255) to point to a location at the bottom of memory. This location contains the address of the program that handles that particular interrupt. Such an address is called an *interrupt vector.*

IBM uses some of the 256 interrupts for the PC's built-in operating system (called its Basic I/O system, or *BIOS*), Disk Operating System (DOS), and BASIC. Others are available for your use. We'll discuss interrupts in detail in Chapter 6.

Data and Address Buses

The 8088 is a 40-pin integrated circuit, or "chip." Memory addresses come out of the chip on a *20-line address bus.* The first eight lines of the address bus also carry data to and from the microprocessor; they form the 8088's *8-bit data bus.* Thus, we say the address bus and the data bus are "multiplexed." (The four high-order address lines are also multiplexed; they carry status information during memory and I/O operations.)

1.4 Internal Registers

Since this book is primarily devoted to programming the 8088, the most logical place to begin is by discussing the internal registers at your disposal. Figure 1-2 shows the three groups of data and address registers, the 16-bit Instruction Pointer (IP), and the 16-bit Flags register.

Data Registers

You may treat the data registers as either four 16-bit registers or eight 8-bit registers, depending on whether you are operating on 16-bit words or 8-bit bytes. The 16-bit registers are named AX, BX, CX, and DX. Within these "X" registers are 8-bit registers named AL, AH, BL, BH, CL, CH, DL, and DH; here, L and H identify the low-order and high-order bytes of the 16-bit registers. For example, AL and AH form the low and high bytes of AX, respectively.

All data registers are available for general programming use, but certain instructions also use them implicitly. Specifically:

- *AX, the Accumulator,* is used in word-size multiplication, division, and I/O operations, and in some string operations. The *AL register* is used in the byte-size counterparts of these same operations, and in translate and decimal arithmetic operations. The *AH register* is also used in byte-size multiplications and divisions.
- *BX, the Base register,* is often used to address data in memory.

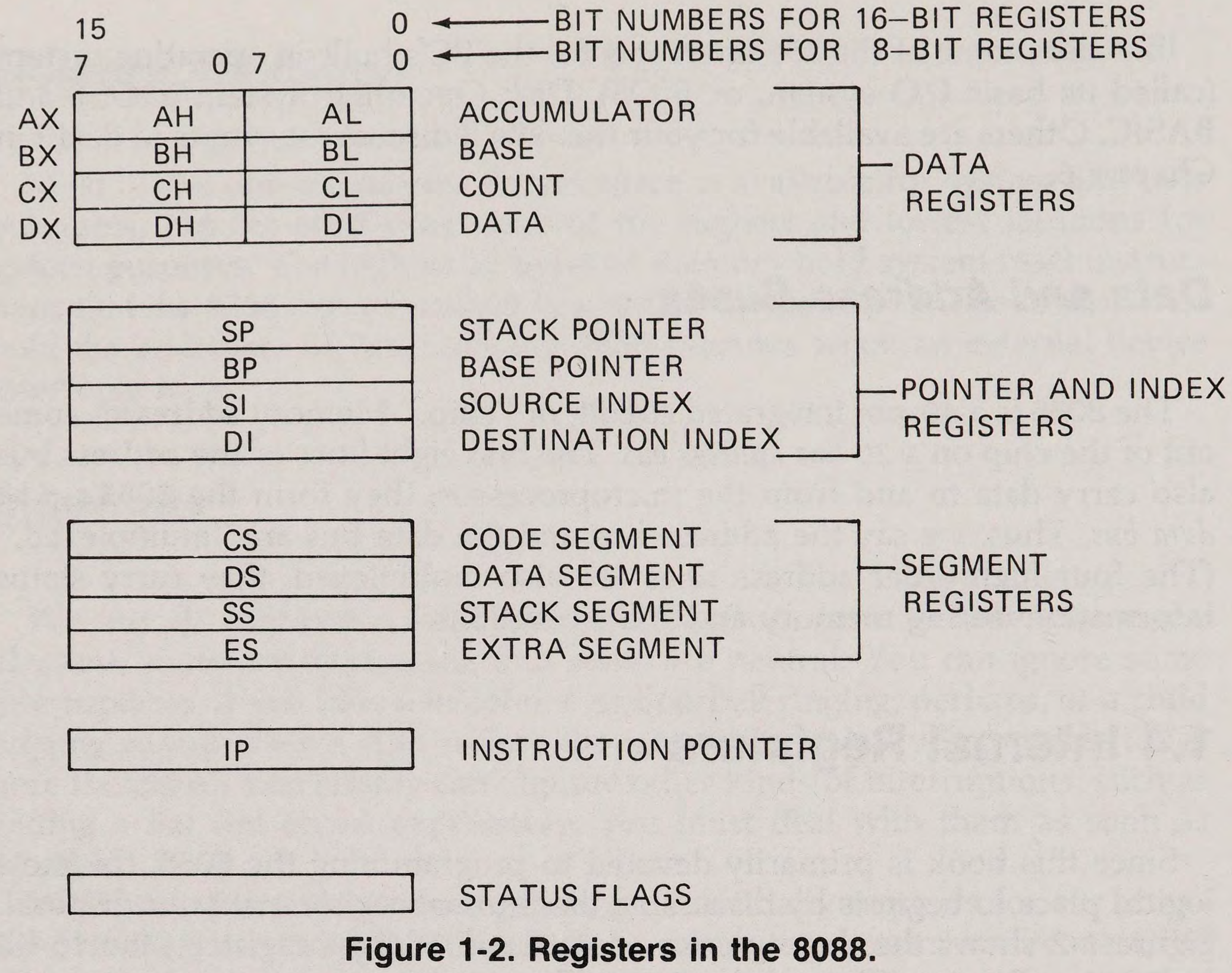


Figure 1-2. Registers in the 8088.

- *CX, the Count register,* acts as a repetition counter for loop operations and as an element counter for string operations. The *CL register* holds the shift count for multiple-bit shift and rotate operations.
- *DX, the Data register,* is used in word-size multiplication and division operations. It can also provide the port number in I/O operations.

Former 8080 or 8085 programmers will note that AH is the only data register unique to the 8088; the others are relabeled 8080/8085 registers. In the 8080, AL is called A, while BX, CX, and DX are called HL, BC, and DE, respectively.

The data registers are the only ones you may reference as either 8- or 16-bit. The registers in the remaining groups are exclusively 16-bit registers.

Segment Registers

As we mentioned earlier, 8086- and 8088-based computers keep programs and data in separate areas of memory. These areas or *segments* can be up to 64K bytes long. The 8088 can work with up to four segments at a time; it holds the starting address of these segments in four *segment registers.* The functions of the segment registers are:

- The *code segment (CS) register* points to the segment that holds the program currently being executed. The 8088 combines the contents of CS (multiplied by 16) with the contents of the instruction pointer (IP) to calculate the memory address of its next instruction.
- The *stack segment (SS) register* points to the current stack segment. A stack is a data structure in memory that acts as a temporary depository for data and addresses. The 8088 uses the stack to hold a return address while a subroutine is being executed, but you may also use the stack to preserve the contents of registers that a subroutine alters. We will say more about stacks later.
- The *data segment (DS) register* points to the current data segment, which usually holds variables.
- The *extra segment (ES) register* points to the current "extra" segment, which is used in string operations.

Pointer and Index Registers

Remember that the 8088 combines a segment number in CS with an offset in IP to calculate the address of an instruction in the code segment. It accesses data in other segments similarly, by combining a segment number in a segment register with an offset in another register. To access the data segment, it gets the segment number from DS and the offset from BX or an *index register (SI or DI).* To access the stack segment, the 8088 gets the segment number from SS and the offset a *pointer register (SP or BP).* It can also access the extra segment by using a segment number from ES (more on this in Chapter 2).

Instruction Pointer

Most microprocessors execute a program by fetching an instruction from memory, executing it, then fetching the next instruction, and so on. This plodding, one-at-a-time approach naturally introduces some delay, because the microprocessor must wait until each new instruction has been fetched from memory before executing it. The 8088 eliminates much of this delay by assigning these two tasks—fetching instructions and executing instructions—to separate, special-purpose units within the chip.

One of these, the *Bus Interface Unit (BIU),* fetches instructions from memory and passes data between the execution hardware and the "outside world." The other, the *Execution Unit (EU),* only executes instructions. Because these units are independent, the BIU can fetch a new instruction from memory at the same time the EU is executing a previously-fetched instruction.

Whenever the BIU fetches an instruction, it puts it in a *pipeline* within the microprocessor. (A pipeline is simply the electronic equivalent of a cigarette vending machine.) Thus, when the EU finishes executing a program instruction, it can usually find the next one in the pipeline.

Since the BIU cannot know the sequence in which the program will execute, it always fetches instructions from consecutive memory locations. Therefore, the only time the EU must wait for an instruction to be fetched is when program execution transfers to a new, non-sequential address. When that happens, the EU must wait for the BIU to clear the pipeline and fetch the next instruction. Then, and only then, the 8088 waits like many other microprocessors wait for *every* instruction to be fetched. Figure 1-3 shows parallel fetching and executing in the 8088.

Because the 8088 works in this rather unique way, the people at Intel chose to differentiate their "next execution address" register from other manufacturers' "next fetch address" register by calling it an *instruction pointer (IP)* instead of a program counter (PC). The IP always contains the offset of the instruction the 8088's EU will execute next. Because the IP has this one dedicated purpose, you cannot perform arithmetic on its contents. However, the 8088 has instructions that explicitly change the IP and others that transfer its contents to and from the stack.

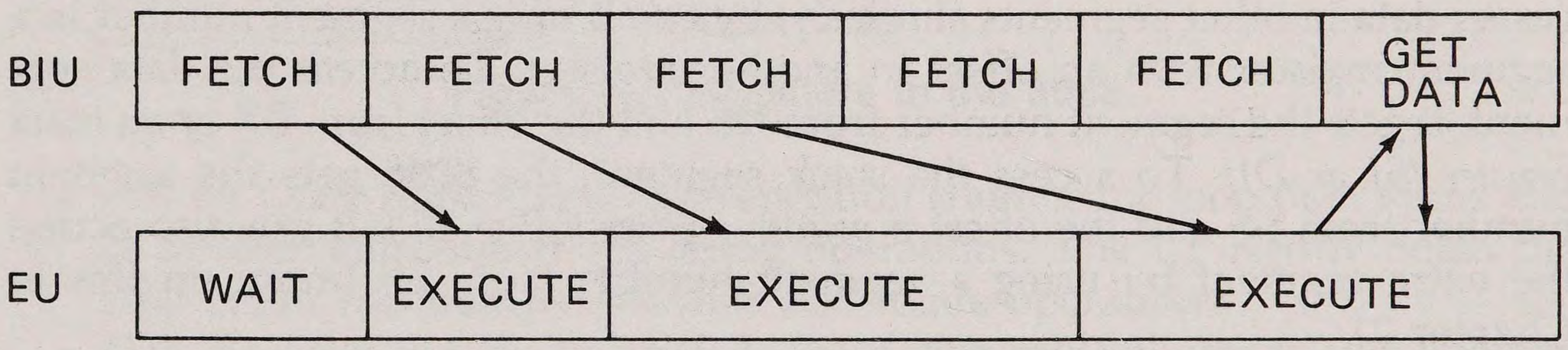


Figure 1-3. Parallel operation in the "pipelined" 8088.

Flags

You will often want your program to make a "decision" based on the result of the instruction the 8088 just executed. For example, you may want to do one thing if an addition yields zero (perhaps print "Balance is zero!" in an accounting program) and something entirely different otherwise.

The 16-bit *Flags register* reports various status conditions that help your programs make decisions. Six bits hold statuses; three others let you control the 8088 from within a program.

Figure 1-4 shows how these nine "flags" are arranged in the Flags register. The flags operate as follows:

1. *Bit 0, the Carry Flag (CF),* is one if an addition produces a carry or a subtraction produces a borrow; otherwise, it is zero. CF also holds the

value of a bit that has been shifted or rotated out of a register or memory location, and reflects the result of a compare operation. Finally, CF also acts as a result indicator for multiplications; see the description of bit 11 (OF) for details.

2. *Bit 2, the Parity Flag (PF),* is one if the result of an operation has an even number of one bits; otherwise, it is zero. PF is used primarily in data communications.
3. *Bit 4, the Auxiliary Carry Flag (AF),* is similar to the CF bit, except AF reflects the presence of a carry or borrow out of bit 3. CF is useful for operating on "packed" decimal numbers.
4. *Bit 6, the Zero Flag (ZF),* is one if the result of an operation is zero; a non-zero result clears ZF to zero.
5. *Bit 7, the Sign Flag (SF),* is meaningful only during operations on signed numbers. SF is one if an arithmetic, logical, shift, or rotate operation produces a negative result; otherwise, it is zero. In other words, SF reflects the most-significant (sign) bit of the result, regardless of whether the result is eight or 16 bits long.
6. *Bit 8, the Trap Flag (TF),* makes the 8088 "single-step" through a program, for debugging purposes.
7. *Bit 9, the Interrupt Enable Flag (IF),* allows the 8088 to recognize interrupt requests from external devices in the system. Clearing IF to zero makes the 8088 ignore interrupt requests (until IF becomes one).
8. *Bit 10, the Direction Flag (DF),* makes the 8088 decrement (DF = 1) or increment (DF = 0) the index register(s) after executing a string instruction. If DF is zero, the 8088 progresses forward through a string (toward higher addresses, or "left to right"). If DF is one, it progresses backward (toward lower addresses, or "right to left").
9. *Bit 11, the Overflow Flag (OF),* is primarily an error indicator during operations on signed numbers. OF is one if the addition of two like-signed numbers, or the subtraction of two opposite-signed numbers, produces a result that exceeds the capacity of the operand; otherwise, it is zero.

   OF is also one if the most-significant (sign) bit of the operand changed during an arithmetic shift operation; otherwise, it is zero.

   The OF flag, in combination with the CF flag, also indicates the length of a multiplication result. If the upper half of the product is non-zero, OF and CF are one; otherwise, both flags are zero.

   Finally, OF is zero if the quotient produced by a divide operation overflows the result register.

The shaded bit positions in Figure 1-4 (1, 3, 5, and 12 through 15) are unused. If you ever read the status of the Flags register, these bits will be zero.

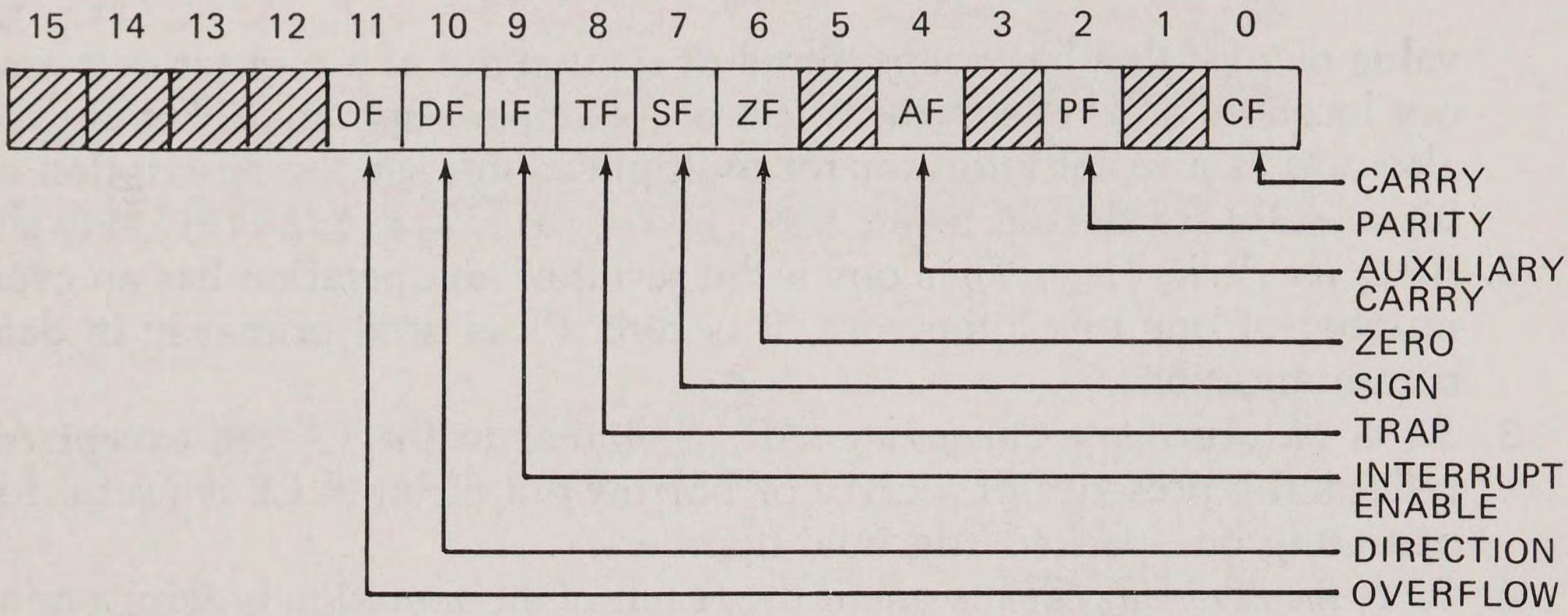


Figure 1-4. Flags register.

Don't make the mistake of assuming that the flags are in one state or another at any given time. The 8088 has instructions that let you set or reset flags. When in doubt, *use* these instructions.

The 8088 has conditional transfer instructions that test the state of the Carry (CF), Parity (PF), Zero (ZF), Sign (SF), and Overflow (OF) flags, and cause program execution to continue in-line or at some other location in memory, depending on the outcome of the test.

Study Exercises (answers on page 361)

1. How does the instruction set of the 8088 differ from that of the 8086?
2. What physical address does the 8088 generate when you combine an offset value of 2H with a segment register containing 4000H?
3. If the AX register contains 1A2BH, what does AL contain?
4. Which segment register normally accesses variables in your programs?
5. Which bit in the Flags register tells whether a preceding subtract operation produced a negative result?

2

Using an Assembler

2.1 Introducing the Assembler

Assembly language provides the best of two worlds. It lets you write programs at the level the microprocessor understands, but doesn't force you to memorize a set of numeric codes. Instead, you write instructions as English-like abbreviations, then run an *assembler* program to convert the abbreviations to their numeric equivalents.

We refer to the program written using abbreviations as the "source program" and the numeric, microprocessor-compatible form of it as the "object program." Therefore, the assembler's job is to convert source programs *you* can understand into object programs the *microprocessor* can understand.

The IBM Macro Assembler

Rather than trying to describe every assembler on the market, we will concentrate on just one popular package: IBM's *Macro Assembler*. The features this package provides should be similar to those of any other assembler you might own. To date, IBM has produced two versions of the Macro Assembler, 1 and 2. Both versions operate essentially the same, but Version 2 supports DOS 2 pathnames, recognizes instructions for an 8087 Math Coprocessor chip and for an Intel 80286 microprocessor (used in the IBM PC AT), and has several other important features that we will describe in this book.

The Macro Assembler disk holds two separate assembler programs, called the *Small Assembler (ASM)* and the *Macro Assembler (MASM).* The Small Assembler only requires 64K bytes of memory (96K with assembler Version 2), while the Macro Assembler requires 96K bytes (128K if you are using Version 2 with DOS 2.0). A major difference is that the Macro Assembler lets you define "macros"—instruction sequences you reference by a single word in a program—whereas the Small Assembler does not. It also produces read-

able error messages; the Small Assembler only produces error codes. Finally, the Macro Assembler supports the 8087 and the 80286; the Small Assembler does not.

Since most PCs nowadays have at least 128K bytes, we will concentrate on the Macro Assembler, but mention where it differs from the Small Assembler. The IBM *Macro Assembler* manual provides complete details of both assembler programs, so we will not attempt to describe everything. Instead, we will cover only the features you will probably use most often, and provide some tables for quick reference.

2.2 Developing an Assembly Language Program

Although assembly language programs look quite different from BASIC programs, you follow the same procedures to develop them. In assembly language, however, the *mechanics* are more involved. There are six steps in developing an assembly language program:

1. Define the task and design the program. This often requires drawing a *flowchart,* a "blueprint" of how the program operates.
2. Type the program instructions into the computer using an *editor.* This is easiest if you first type general comments in plain English telling what you want to do. Then, between the comments, you insert the instructions that carry out the task.
3. Assemble the program using an *assembler.* If the assembler finds errors, correct them with the editor and reassemble the program.
4. Convert the assembler output to an executable "run module" using the *linker.*
5. Execute (run) the program.
6. Check the results. If they differ from what you expected, you must find the errors or "bugs"; that is, you must "debug" the program.

If your program is short and simple, you should be able to perform these steps quickly. However, longer and more complex programs will require more time on each step; especially defining the program itself. We will suggest an efficient approach for developing programs in the "Top-Down Program Design" section at the end of this chapter.

Editor

Step 2 refers to an *editor.* This is a program that lets you enter and prepare your program. You can use any popular word processor or editor program

that can produce pure "ASCII" text—regular characters without any special control codes or formatting codes. These include *WordStar, Microsoft Word, MultiMate,* and IBM's *Personal Editor.*

If you don't have one of those programs, you can use the *EDLIN* program that comes on the DOS disk. EDLIN is a line editor; that is, its commands operate on numbered lines in your program. You will learn how to use it in Section 2.7.

Assembler

The computer cannot directly execute the program you prepare using an editor. An assembler must convert it into an *object program* the computer can understand.

Linker (LINK)

The IBM DOS can store a program at any convenient place in memory, which frees you from having to tell it where to put the program. However, to use this feature you must convert the assembled program to a form that can be moved around (the computer term is *relocatable*). This involves using a program called LINK. LINK is on the DOS Supplemental Programs disk. However, Version 2 of the Macro Assembler contains an enhanced version (2.20) for users of DOS 2.0 and higher; this is the LINK that comes with DOS 3.0.

The IBM manuals refer to a disk file that contains an assembled program as an *object module.* Similarly, they refer to a file that contains a relocatable version of the assembled program as a *run module.* Hence, LINK's job is to create a run module from an object module.

A Second Job for the Linker

You can also build programs one section at a time, like you would build a model airplane. To do this, you enter one section of the program and assemble it. If the assembler reports errors, you correct them and assemble again. Once this first module assembles without errors, you repeat the process for the second module, then for the third (if any), and so on. Thus, you eventually wind up with several object modules that, in combination, do whatever you designed the program to do.

You can probably guess what the Linker's second job is. It links assembled object modules to form a single run module.

You must run the Linker for every program you write, even those that have only one object module. For programs with one module, the Linker simply makes

the module relocatable. For programs with two or more modules, it combines them and then makes the *result* relocatable.

A Debugging Program (DEBUG)

You can run your completed program in two ways:

1. You can simply enter its name from DOS.
2. You can run it under control of the DEBUG program.

Generally, you only run a program from DOS when you are sure it has no mistakes. Until then, you run it under DEBUG.

The DEBUG program (on the DOS Supplemental Programs disk) lets you control your program as it runs. Among other things, DEBUG lets you change and display values, stop your program at a particular point, or run it one instruction at a time. DEBUG thus provides you with tools that help to identify and correct errors in programs.

"Top-Down" Program Design

When creating a program on a computer, one's natural inclination is to charge into it just as you would with pencil and paper: to enter the first instruction, then the second, then the third, and so on, to the end. This "brute force" approach may work for programs that are short or simple, but it generally leads to errors and produces programs that are difficult to understand (and, as a result, even more difficult to update later). Thanks to the editing capabilities of word processor and editor programs, there is an easier, more reliable, and more efficient way to develop programs. It is called *top-down design.*

Top-down design simply means starting with a plain English outline of the program, then gradually filling in the details. The outline should be a series of lines that tell what steps you want the program to perform. For example, to develop a program that performs one of several tasks based on a choice the user makes from a menu, your outline might look like this:

```
; Display a menu of selections.
; Ask the user to choose.
; Read user's selection.
; Check for an illegal entry.
; If entry is legal, do what the user has asked.
```

The semicolons here indicate that these lines represent comments rather than instructions. They do the same thing as REMs in BASIC.

From here, you can use the editor to insert instructions between comment lines. Because each line defines a small task, an easy approach is to complete them individually, and test each one before proceeding further. That is, begin by inserting the first group of instructions (the ones that display a menu, in our example), then save the program on disk and perform the rest of the program steps (assemble, link, and execute). Executing this partially-completed program will tell you whether it is working correctly so far. If it isn't, debug it and try again. When the first part is working correctly, proceed to the second part, then the third, and so on.

This may seem like an awfully slow way to develop a program, but it has several advantages:

1. It forces you to plan an orderly approach to the program.
2. The comment lines provide a certain amount of top-level documentation to the finished product.
3. It ensures that each step is working correctly before you proceed.

2.3 Source Statements

Now that we have discussed the mechanics of developing programs, we can finally look at what goes into one. The program you write—the source program—is a sequence of *statements* designed to perform a specific task. A source statement (a line in the program) can be either an assembly language instruction or a pseudo-operation.

Assembly language instructions are shorthand notations for the 8088 microprocessor's instruction set. Some manuals refer to them as "machine instructions," because they tell the "machine" (the 8088) what to do. By contrast, *pseudo operations*—or "pseudo-ops" for short—tell the *assembler* what to do with the instructions and data you enter.

Source statements of either kind can also include *operators,* which give the assembler information about an operand, where ambiguities exist. We discuss assembly language instructions, pseudo-ops, and operators in the sections that follow.

Constants in Source Statements

The assembler lets you enter constants in several forms. The most common are:

1. *Binary*—A sequence of 1's and 0's followed by the letter B; for example, 10111010B.

2. *Decimal*—A sequence of the digits 0 through 9, with or without the letter D; for example, 129D or 129.
3. *Hexadecimal*—A sequence of the digits 0 through 9 and the letters A through F, followed by the letter H. The first character must be one of the digits 0 through 9; for example, 0E23H. (Here, the 0 prefix tells the 8088 that E23H is a *number,* rather than a symbol or a variable name.)
4. *Character*—A string of letters, numbers, or symbols enclosed in single or double quotes. IBM provides both forms because you sometimes want to put quotation marks inside message strings, as in "Don't enter a number here!" or "Your reply to the 'Please try again' prompt is illegal."

Negative Numbers

You can also enter negative numbers. If the number is a decimal value, you simply precede it with a minus sign (e.g., -32). If it is binary or hexadecimal, you must enter its "two's-complement" form. For example, 11100000B and 0E0H are two's-complement forms of decimal -32.

2.4 Assembly Language Instructions

Each assembly language instruction in a source program can have up to four *fields,* as follows:

```
[Label:] Mnemonic [Operand] [;Comment]
```

Of these, only the mnemonic field is always required. The label and comment fields are always optional. The operand field applies only to instructions that *require* an operand; otherwise you must omit it. (We show the label, operand, and comment fields in brackets here to identify them as optional; *don't* type the brackets into your programs.)

You may enter these fields anywhere on a line, but you must separate them with at least one space (or tab). An assembly language instruction that uses all four fields is:

```
GETCOUNT: MOV CX,DI    ;Initialize count
```

Label Field

The label field assigns a *name* to an assembly language instruction. This lets other instructions in the program refer to the instruction. Note that labels

in assembly language programs serve the same purpose as line numbers in BASIC programs.

An instruction label can be up to 31 characters long and must end with a colon (:). It may consist of:

- The letters *A* through *Z* or *a* through *z* (the assembler doesn't distinguish between lowercase and uppercase)
- The numeric digits *0* through *9*
- These special characters: ? . @ _ $

You can start a label with any character except a digit, but if you use a period (.), it must be the first character. The symbols AH, AL, AX, BH, BL, BX, BP, CH, CL, CX, CS, DH, DL, DX, DI, DS, ES, SI, SP, and ST are register names; you can't use them as labels. Nor can you use instruction names or *mnemonics* (listed in Chapter 3) as labels.

You can't put a space in a label, but you can get the same effect by using an underscore character (_). For example, you could write the previous sample statement as

```
GET_COUNT: MOV CX,DI   ;Initialize count
```

Clearly, GET_COUNT is more readable than GETCOUNT.

Selecting Label Names

Because the assembler lets you enter various combinations of letters, digits, and symbols, almost any label you can think of is acceptable. However, we recommend the following guidelines for selecting labels:

- Make the name as short as possible, while still being reasonable. Thus, MPH is preferable to MILES_PER_HOUR and CUR_YR is a reasonable abbreviation for CURRENT_YEAR.
- Make the name easy to type without errors. The usual typing problems are several identical letters in a row (such as HHHH) and similar-looking characters (such as the letter O and the number 0, letter I and number 1, and letter S and number 5). There is no reason to invite typing errors; most of us make enough of them anyway.
- Do not use labels that could be confused with each other. For example, avoid using things like XXXX and XXYX. There's no sense in tempting fate and Murphy's Law.

Mnemonic Field

The mnemonic field (the leading "m" in mnemonic is silent) holds the two- to six-letter acronym for the microprocessor instruction. For example, MOV is the acronym for a move instruction and ADD is the acronym for an add instruction. The assembler uses an internal table to translate each mnemonic in the program into its numeric equivalent.

In addition to the mnemonic, many instructions require you to specify either one or two operands. The mnemonic tells the assembler how many operands, and which types, to obtain from the operand field. We describe the legal mnemonics in Chapter 3.

Operand Field

The operand field tells the 8088 where to find the data it is to operate on. For example, in this *move* instruction:

```
MOV CX,DX
```

CX,DX tells it to copy the contents of the DX register into the CX register.

The operand field is mandatory with some instructions and prohibited with others. If present, the operand field will contain either one or two operands, separated from the mnemonic by at least one space or tab. If two operands are required, you must put a comma between them.

For two-operand instructions, the first is the *destination operand* and the second is the *source operand.* The source operand specifies the value that the microprocessor should add to, subtract from, compare to, or store into the destination operand. (The preceding MOV example tells it to store source operand DX into destination operand CX.) For this reason, the source operand is never altered by the operation, whereas the destination operand is nearly always altered. In Chapter 3 we will discuss the addressing characteristics for each instruction in the 8088's instruction set.

Comment Field

Like a REM in BASIC, this optional field lets you describe statements in the source program to make the program easier to understand. You must precede a comment with a semicolon (;) and separate it from the preceding field by at least one space or tab. The assembler ignores comments, but prints them when you list the program.

Comments should describe what is happening in the program, not just restate the instruction. For example,

```
MOV CX,0  ;Clear the count register
```

is more meaningful than

```
MOV CX,0  ;Move 0 into CX
```

Stand-Alone Comments

You may also use comments by themselves to introduce a program or a portion of one. To put a stand-alone comment in a program, enter a semicolon (;) at the beginning of the line. The assembler will recognize the semicolon as the start of a comment line, and ignore whatever follows it.

2.5 Pseudo Operations

Pseudo operations, or *pseudo-ops,* are commands to the assembler rather than to the microprocessor. Pseudo-ops can set up segments and procedures (i.e., subroutines), define symbols, reserve memory for temporary storage, and perform a variety of other important "housekeeping" tasks. Unlike assembly language instructions, however, most pseudo-ops generate no object code.

Pseudo-op statements can have up to four fields as shown here:

```
[Name] Pseudo-op [Operand] [;Comment]
```

As the brackets indicate, only the pseudo-op field is always required. A name is mandatory with some pseudo-ops, prohibited with others, and optional with the rest. The same applies to an operand. The comment field is always optional. As with assembly language instructions, you can put pseudo-op fields anywhere on a line, but you must separate them with at least one space or tab.

The Macro Assembler provides about 60 different pseudo-ops. In this section we discuss the most common ones; we will cover some advanced pseudo-ops in Section 2.8 and macro pseudo-ops in Chapter 9. Table 2-1 divides the common pseudo-ops into two groups: data and listing.

Note: Don't try to memorize the material in this section. Just read it casually, then come back to it later when you want to look up some detail.

Table 2-1. Common pseudo-ops.

Type	*Pseudo-ops*		
Data	ASSUME	END	EXTRN
	COMMENT	ENDP	INCLUDE
	DB	ENDS	PROC
	DW	EQU	PUBLIC
	DD	= (Equal Sign)	SEGMENT
Listing	PAGE	SUBTTL	TITLE

Data Pseudo-Ops

We can divide the assembler's data pseudo-ops into five functional groups, as shown in Table 2-2.

Table 2-2. Data pseudo-ops.

Pseudo-Op	*Function*
Symbol Definition	
EQU	Format: `name EQU text` or `name EQU numeric-expression` Assigns *text* or value of *numeric-expression* to *name,* permanently.
=	Format: `name = numeric-expression` Assigns value of *numeric-expression* to *name,* but can be reassigned.
Data Definition	
DB	Format: `[name] DB expression[,...]` Defines a variable or initializes storage. DB allocates one or more bytes.
DW	Format: `[name] DW expression[,...]` Similar to DB, but allocates two-byte words.
DD	Format: `[name] DD expression[,...]` Allocates four-byte doublewords.
External Reference	
PUBLIC	Format: `PUBLIC symbol[,...]` Makes the defined *symbol*(s) available for use by other assembly modules that will be linked to this module.

Table 2-2. Data pseudo-ops (continued).

Pseudo-Op	*Function*
EXTRN	Format: `EXTRN name:type[,...]` Specifies symbols defined in another assembly module.
INCLUDE	Format: `INCLUDE filespec` Merges the contents of the specified source file into the current source file.
Segment/Procedure Specification	

SEGMENT	Format:
	`seg-name SEGMENT [align-type] [combine-type] ['class']`
	`..`
	`..`
	`seg-name ENDS`
	Defines the boundaries of a segment. Each *SEGMENT* definition must end with an *ENDS* statement.

ASSUME	Format:
	`ASSUME seg-reg:seg-name[,...]`
	or
	`ASSUME seg-reg:NOTHING[,...]`
	Tells the assembler which segment register (CS, DS, ES, or SS) a segment belongs to. *ASSUME NOTHING* cancels any previous ASSUME for the specified register.

PROC	Format:
	`name PROC [NEAR]`
	or
	`name PROC FAR`
	`..`
	`..`
	`RET`
	`name ENDP`
	Assigns a *name* to a sequence of assembler statements. Every *PROC* definition must end with an *ENDP* statement.

Assembly Control	
END	Format: `END [entry-point label]` Marks the end of the source program.

Symbol Definition Pseudo-Ops

The symbol definition pseudo-ops let you assign a symbolic *name* to an *expression*. The expression may be a 16-bit constant, an address reference, another symbolic name, a segment prefix and operand, or an instruction la-

bel. After assigning the name, you can use it anywhere you would normally use the expression.

The EQU (equate) and = (equal sign) pseudo-ops are similar, but:

1. You can redefine symbols defined with =, while symbols defined with EQU are permanent.
2. EQU can be used for either text or a numeric expression, while = can be used only for numeric expressions.

EQU is handy for assigning simple names to numbers, complex addressing combinations, and other things you wouldn't want to enter over and over in your programs. Some examples are:

```
K       EQU 1024                ;Name a constant
TABLE EQU DS:[BP][SI]           ;Name an addressing combination
SPEED EQU RATE                  ;Give an alternate name
COUNT EQU CX                    ;Give a register a name
```

Being an expression, the operand can also include some simple math, where you let the assembler make the calculation. For example:

```
DBL_SPEED       EQU 2*SPEED
MINS_PER_DAY    EQU 60*24
```

Some examples of the = pseudo-op are:

```
CONST = 56          ;This is the same as CONST EQU 56, but
CONST = 57          ; now CONST may be redefined, or
CONST = CONST+1     ; may refer to its previous definition
```

Data Definition Pseudo-Ops

Many programs use locations in memory to hold *variables*—named data items that can be changed as needed. The pseudo-ops we commonly use to allocate space for variables are *DB* (Define Byte), *DW* (Define Word), and *DD* (Define Doubleword).

These differ primarily in the amount of memory they allocate. DB allocates 8-bit bytes, DW allocates 2-byte words, and DD allocates 4-byte doublewords. When defining a variable, you can either give it a specific initial value or simply reserve the space and insert the value later.

The data definition pseudo-ops have the general formats

```
[name] DB expression[,...]
[name] DW expression[,...]
[name] DD expression[,...]
```

where the operand *expression* can take any of several forms, depending on how you want to define the variable.

For instance, the expression may be a *constant.* The following statements show the allowable maximum and minimum decimal values for byte- and word-size variables:

```
BU_MAX DB 255        (maximum byte constant, unsigned)
BS_MAX DB 127        (maximum byte constant, signed)
BS_MIN DB -128       (minimum byte constant, signed)

WU_MAX DW 65535      (maximum word constant, unsigned)
WS_MAX DW 32767      (maximum word constant, signed)
WS_MIN DW -32768     (minimum word constant, signed)
```

You can also use pseudo-ops to set up data tables in memory. To do this, simply list the table elements separated by commas. The following sequence sets up two 12-element tables, one made up of bytes and the other of words:

```
B_TABLE   DB  0,0,0,0,8,-13                    (byte table)
          DB  -100,0,55,63,63,63
W_TABLE   DW  1025,567,-30222,0,90,-129        (word table)
          DW  17,645,26534,367,78,-17
```

Here we have arranged the elements as two lines of six values, but you may assign *any* number of values with one pseudo-op, as long as you don't put more than 132 characters on a line.

Note that the first four elements and the last three elements of B_TABLE have the same value (0 and 63, respectively). The assembler has a *DUP* (duplicate) operator that allows you to repeat operands without entering them individually. Using DUP, we could set up B_TABLE with the shorter statement

```
B_TABLE   DB  4 DUP(0),8,-13,-100,0,55,3 DUP(63)
```

To define a variable without giving it an initial value, put a question mark (?) in the expression field. For example, the following statements reserve space for a byte and a word in memory, but don't store anything into them:

```
HIGH_TEMP    DB ?
AVG_WEIGHT   DW ?
```

Note that the ? simply *reserves* space for HIGH_TEMP and AVG_WEIGHT; it does not *initialize* them in any way! Don't make the mistake of assuming they contain 0 or any other specific value.

You may also reserve space for an entire table. For example,

```
MONTHLY_SALES   DW   12 DUP(?)
```

reserves 12 words in memory. This does roughly the same thing as *DIM MONTHLY_SALES%(12)* in BASIC.

The DB (Define Byte) pseudo-op can also accept a *character string* as an expression. This lets you store error messages, table headers, and other text in memory. In this case, you must enclose the string in single or double quotes. Two examples are:

```
POLITE_MSG   DB   'The number you have entered is too large'
             DB   ' to be properly processed. Please'
             DB   ' re-enter your data.'
RUDE_MSG     DB   'Try again, dummy!'
```

Variables are also useful for holding *memory addresses* that can be referenced by instructions in your program. As you know, each address has two components: a segment number and an offset.

If a label lies in the same segment as the instruction that refers to it, you can specify only its offset. Since an offset is 16 bits long, we can use DW to hold it. For example, the statement

```
HERE_NEAR   DW   HERE
```

assigns the 16-bit offset of the label HERE to the name HERE_NEAR. We refer to a variable that holds an offset as a *pointer.*

If a label lies in a different segment than the instruction that refers to it, the 8088 must know the label's segment number as well as its offset. You can provide both components with a DD (Define Doubleword) pseudo-op. For example,

```
HERE_FAR   DD   HERE
```

assigns the 16-bit offset and the 16-bit segment number of the label HERE to the 32-bit variable HERE_FAR. A variable that holds both components is called a *vector.*

Segment/Procedure Specification Pseudo-Ops

The SEGMENT and ENDS pseudo-ops divide the source program into segments. As we mentioned earlier, a program may have up to four kinds of segments: data, code, extra, and stack.

For example, a data segment may look like this:

```
DATASEG  SEGMENT  PARA PUBLIC 'DATA'
  A          DB  ?
  B          DB  ?
  SQUARES    DB  1,4,9,16,25,36,49,64
DATASEG  ENDS
```

and a code segment may look like this:

```
PROGCODE  SEGMENT  PARA PUBLIC 'CODE'
          ..
          ..
          MOV      AX,BX
          MOV      CL,DH
          MOV      DI,CX
          ..
          ..
PROGCODE  ENDS
```

The words SEGMENT and ENDS simply mark the beginning and end of a segment; they don't tell the assembler which *kind* of segment is being defined. A separate pseudo-op, ASSUME, does that.

ASSUME has the general format

```
ASSUME  seg-reg:seg-name[,...]
```

where *seg-reg* is either DS, CS, SS, or ES and *seg-name* is the name that precedes a SEGMENT pseudo-op. DS points to the data segment, CS to the code segment, SS to the stack segment, and ES to the extra segment.

ASSUME helps the assembler translate labels into addresses, by telling it which segment register you plan to use to address those labels. For example, the statement

```
ASSUME  DS:DSEG
```

says to the assembler, "The segment called DSEG is my data segment. When you assemble my program and find mention of a label in DSEG, tell the

microprocessor to obtain the label's segment number from DS. In turn, *I* promise to make DS point to the beginning of DSEG."

We normally put the ASSUME immediately after the code segment's SEGMENT statement. Thus, for the preceding two-segment program, the code segment would have this form:

```
PROGCODE  SEGMENT  PARA PUBLIC 'CODE'
          ASSUME   CS:PROGCODE,DS:DATASEG
          MOV      AX,DATASEG  ;Make DS point to DATASEG
          MOV      DS,AX
          ..
          ..
          MOV      AX,BX
          MOV      CL,DH
          MOV      CX,DI
          ..
          ..
PROGCODE  ENDS
```

Again, note that we must explicitly load the data segment's address into DS; the ASSUME does not take care of this.

The PROC and ENDP pseudo-ops mark the beginning and end of a *procedure.* A procedure is a set of instructions that can be executed from various places in a program. Whenever your program calls a procedure, the 8088 executes it, then returns to the place where the call was made. Because you write a procedure just once in a program, it frees you from typing the sequence each time you need it, and thereby makes your programs shorter.

Every procedure must begin with a PROC and end with an ENDP. If it also contains a RET (Return from Procedure) instruction—and most do—we can call it a *subroutine.* The RET instruction makes the 8088 resume at the point where the subroutine was called, just as RETURN does in a BASIC subroutine.

A procedure always has one of two *distance* attributes: NEAR or FAR, as specified by the operand that follows PROC. Omitting the operand makes the procedure NEAR.

A NEAR procedure can only be called from within the code segment in which it is defined. For example:

```
CSEG   SEGMENT  PARA PUBLIC 'CODE
       ASSUME   CS:CSEG
CALLER  PROC
          ..
          CALL   CALLEE    (Call to a procedure)
          ..
          RET
CALLER  ENDP
```

```
CALLEE   PROC NEAR        (Procedure that is called)
         ..
         ..
         RET
CALLEE   ENDP
CSEG   ENDS
```

Here, the called procedure (CALLEE) happens to be defined in the same segment as the CALL instruction. However, the two procedures may also be in different program modules (i.e., different disk files), as long as their code segments have the same name. Communicating between modules involves the EXTRN and PUBLIC pseudo-ops, which we'll discuss in the next section.

A FAR procedure can be called from any code segment. For example:

```
CSEG   SEGMENT   PARA PUBLIC 'CODE
       ASSUME    CS:CSEG
CALLER  PROC
         ..
         CALL   CALLEE   (Call to a procedure)
         ..
         RET
CALLER   ENDP
CSEG   ENDS

CSEG1   SEGMENT   PARA PUBLIC 'CODE'
        ASSUME    CS:CSEG1
CALLEE   PROC   FAR    (Procedure that is called)
         ..
         ..
         RET
CALLEE   ENDP
CSEG1   ENDS
```

When the 8088 calls a procedure, it pushes a return address onto the stack. It will retrieve this address when it executes the RET instruction. If the procedure has a NEAR attribute, the call puts only an offset—the contents of the instruction pointer (IP)—on the stack. If the procedure has a FAR attribute, the call puts both a segment number—the contents of the code segment (CS) register—and an offset from IP on the stack (in that order).

When you assemble the program, the assembler translates each RET instruction into machine code that tells the 8088 how many return address words to retrieve from the stack. A RET in a NEAR procedure makes it remove only one word (IP contents) from the stack; a RET in a FAR procedure makes it remove two words (IP and CS contents).

All this NEAR and FAR business may have you totally confused. Here are some rules that will help you decide whether to make your procedures NEAR or FAR:

1. You can run a program using either DOS or DEBUG. However, since both DOS and DEBUG are in different code segments than your program, *your main procedure must be FAR.*
2. If you always give your code segments the same name (e.g., CSEG), make all procedures NEAR except the main one.
3. If you are using someone else's programs (e.g., the programs on the disk that supplement this book), assume their procedures are FAR. We will discuss how to do this shortly, when we discuss the EXTRN pseudo-op.

External Reference Pseudo-Ops

These pseudo-ops allow you to use information in an assembly module or file that is elsewhere in the system.

The PUBLIC pseudo-op makes the specified symbol(s) available to other assembly modules that will eventually be linked to the present module. A symbol can be a variable name, a label (including PROC labels), or a name defined by an EQU or = pseudo-op.

The EXTRN pseudo-op identifies symbols that are defined (and declared PUBLIC) in some other assembly module. EXTRN has the general form

```
EXTRN  name:type[,...]
```

where *name* is the symbol defined in the other assembly module and *type* can be one of the following:

- If *name* is a symbol in a data segment or extra segment, then *type* can be BYTE, WORD, or DWORD.
- If *name* is a procedure label, then type can be NEAR or FAR.
- If *name* is a constant defined by an EQU or = pseudo-op, then *type* must be ABS.

For example, suppose you want to access a variable called TOTAL that is in another module. The module in which TOTAL is defined may include these statements:

```
        PUBLIC  TOTAL
TOTAL   DW      0               ;Make total = 0 to start
```

and the module in which TOTAL is referenced would contain this statement:

```
EXTRN  TOTAL:WORD
```

The INCLUDE pseudo-op merges (reads) an entire *file* of source statements into the current source file at assembly time. For example,

```
INCLUDE  B:OTHERFIL.ASM
```

reads the contents of the file OTHERFIL.ASM on drive B into your source file, replacing the INCLUDE statement. You can also use INCLUDE to read *macros* into a program. We'll say more about this in Chapter 9.

Assembly Control Pseudo-Op

The assembler recognizes several assembly control pseudo-ops, but only END is frequently used. See Section 2.8 for the others.

The END pseudo-op marks the end of the source program, and tells the assembler where to stop assembling. Therefore, *END must be included in every source program.* Its general form is

```
END  [entry-point label]
```

where *entry-point label* identifies your source program. For example,

```
END  MY_PROG
```

marks the end of the program MY_PROG.

Important: If your program consists of several modules, you must label the END in the main module, but omit it from the ENDs in secondary modules. For example, if your main program calls subroutines that are in separate modules, the END at the end of each subroutine module must be unlabeled.

Listing Pseudo-Ops

The PAGE pseudo-op has the general format

```
PAGE  [lines][,columns]
```

where *lines* and *columns* set the length and width of the pages printed during assembly. Lines can range from 10 to 255; columns from 60 to 132 characters. The defaults are 57 lines and 80 characters. For example,

```
PAGE 25,100
```

limits each page to 25 lines vertically and 100 characters horizontally.

Typically, however, you will want to print a 132-column listing on standard 66-line paper, so the normal statement is

```
PAGE  ,132
```

(If you are using standard 8½-inch paper, set the printer to its smallest type size; usually 16.5 characters per inch.)

On the top line of each page the assembler prints a chapter number and a page number, with a dash between them. The assembler increases the page number when a page is full or when it encounters PAGE alone (no operands). It increases the chapter number only when it encounters *PAGE +* (this also resets the page number to 1). All three conditions make the printer advance to the next page.

The TITLE pseudo-op produces a left-justified title on the second line of each page; we use it to give the assembly module's filename and a short description of what the module does. We generally put the TITLE statement at the beginning of a program, but you can actually put it anywhere.

The SUBTTL pseudo-op produces a centered subtitle on the third line of the next page; this usually describes the contents of the page. For example, the beginning of a listing might have these headers:

```
TITLE   COUNT_ALL - Galaxy Census Program
SUBTTL  Venusian Data Segment
```

At the end of the first data segment, you could change the subtitle as follows:

```
SUBTTL  Plutonian Data Segment
PAGE
```

Titles and subtitles may be up to 60 characters long.

Table 2-3 summarizes the listing pseudo-ops.

Table 2-3. Listing pseudo-ops.

Pseudo-Op	*Function*
PAGE	Format: `PAGE  [lines][,columns]` Sets the length and width of the page.
TITLE	Format: `TITLE  text` Specifies a title to be listed on the second line of each page.
SUBTTL	Format: `SUBTTL  text` Specifies a subtitle to be listed on the third line of each page.

2.6 Operators

Note: This is primarily a reference section. If you are a beginner, skip it for now, then come back later if you need to look up details.

An operator is a modifier used in the operand field of an assembly language or pseudo-op statement. There are five kinds of operators:

1. Arithmetic operators
2. Logical operators
3. Relational operators
4. Value-returning operators
5. Attribute operators

Table 2-4 summarizes these operators.

Table 2-4. Operators.

Operator	*Function*
Arithmetic	
+	Format: `value1 + value2` Adds *value1* and *value2*.
-	Format: `value1 - value2` Subtracts *value2* from *value1*.
*	Format: `value1 * value2` Multiplies *value2* by *value1*.
/	Format: `value1 / value2` Divides *value1* by *value2*, and returns the quotient.
MOD	Format: `value1 MOD value2` Divides *value1* by *value2*, and returns the remainder.
SHL	Format: `value SHL expression` Shifts *value* left by *expression* bit positions. **Note:** Do not use SHL with Macro Assembler Version 1.00.
SHR	Format: `value SHR expression` Shifts *value* right by *expression* bit positions. **Note:** Do not use SHR with Macro Assembler Version 1.00.
Logical	
AND	Format: `value1 AND value2` Takes logical AND of *value1* and *value2*.
OR	Format: `value1 OR value2` Takes logical inclusive-OR of *value1* and *value2*.

Table 2-4. Operators (continued).

Operator	*Function*
XOR	Format: `value1 XOR value2` Takes logical exclusive-OR of *value1* and *value2*. **Note:** Do not use XOR with Macro Assembler Version 1.00.
NOT	Format: `NOT value` Reverses the state of each bit in *value*; that is, it takes the one's complement.
Relational	
EQ	Format: `operand1 EQ operand2` True if the two operands are identical.
NE	Format: `operand1 NE operand2` True if the two operands are not identical.
LT	Format: `operand1 LT operand2` True if *operand1* is less than *operand2*.
GT	Format: `operand1 GT operand2` True if *operand1* is greater than *operand2*.
LE	Format: `operand1 LE operand2` True if *operand1* is less than or equal to *operand2*.
GE	Format: `operand1 GE operand2` True if *operand1* is greater than or equal to *operand2*.
Value-Returning	
$	Format: `$` Returns the current value of the location counter.
SEG	Format: `SEG variable` or `SEG label` Returns the segment value of *variable* or *label*.
OFFSET	Format: `OFFSET variable` or `OFFSET label` Returns the offset value of *variable* or *label*.
LENGTH	Format: `LENGTH variable` Returns the length in units (bytes or words) for any variable defined using DUP.
TYPE	Format: `TYPE variable` or `TYPE label` For variables, TYPE returns 1 (BYTE), 2 (WORD), or 4 (DOUBLEWORD). For labels, it returns -1 (NEAR) or -2 (FAR).

Table 2-4. Operators (continued).

Operator	*Function*
SIZE	Format: `SIZE variable` Returns the product of LENGTH times TYPE.
Attribute	
PTR	Format: `type PTR expression` Overrides the type (BYTE or WORD) or distance (NEAR or FAR) of a memory address operand. *type* is the new attribute and *expression* is the identifier whose attribute is to be overridden.

DS:
ES:
SS:
CS:
	Format: `seg-reg:addr-expr`
	or
	`seg-reg:label`
	or
	`seg-reg:variable`
	Overrides the segment attribute of a label, variable, or address expression.

SHORT	Format: `JMP SHORT label` Modifies the NEAR attribute of a JMP target label to indicate it is no farther than +127 bytes or -128 bytes from the next instruction.
THIS	Format: `THIS attribute` or `THIS type` Creates a memory address operand of either distance attribute (NEAR or FAR) or either type attribute (BYTE or WORD) at an offset equal to the current value of the location counter and a segment attribute of the enclosing segment.
HIGH	Format: `HIGH value` or `HIGH expression` Returns the high-order byte of a 16-bit numeric value or address expression.
LOW	Format: `LOW value` or `LOW expression` Returns the low-order byte of a 16-bit numeric value or address expression.

Arithmetic Operators

The arithmetic operators combine numeric operands and produce a numeric result. The most commonly-used arithmetic operators are those that add (+), subtract (-), multiply (*), and divide (/).

The divide operator (/) returns the quotient produced by a divide operation. For example, the statement

```
PI_QUOT  EQU  31416/10000
```

returns the value 3.

The assembler also has a MOD operator that returns the *remainder* of a divide operation. The statement

```
PI_REM  EQU  31416 MOD 10000
```

defines a constant called PI_REM that has the value 1416.

Finally, operators SHL and SHR displace a numeric operand to the left or right. About the only time you need this capability is if you set up "masks" that you'll apply to binary patterns in memory. For example, if you set up a mask with the statement

```
MASK  EQU  110010B
```

the statement

```
MASK_LEFT_2  EQU  MASK SHL 2
```

sets up a new constant with the value 11001000B. Similarly,

```
MASK_RIGHT_2  EQU  MASK SHR 2
```

sets up a new constant with the value 1100B.

Logical Operators

Like SHL and SHR in the preceding section, the logical operators are used primarily to manipulate binary values instead of decimal values. However, the logical operators manipulate individual bits, rather than an entire group of bits.

To draw an analogy, imagine a group of patients seated on a bench at a medical clinic. If the clinic operates in "shift" fashion, the nurse may order

the first three patients to report to examination rooms and ask the remaining patients to shift left. In doing this, the nurse essentially performs an *SHL 3* operation.

Conversely, if the clinic operates in "logical" fashion, the nurse may order just certain patients (perhaps those with broken bones) to report for examination and tell the remaining patients to stay where they are.

The logical operators AND, OR, and XOR combine two operands to produce a result; NOT requires just one operand. Table 2-5 shows how AND, OR, and XOR operate.

AND is convenient to filter, mask, or strip out certain bits. To do this, it sets the result bit to 1 for each bit position in which both operands contain a 1. For any other bit combination, AND clears the result bit to 0.

Table 2-5. AND, OR, and XOR combinations.

		Result		
Operand #1	***Operand #2***	***AND***	***OR***	***XOR***
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

For example, the operation

```
00110100B AND 11010111B
```

produces 00010100B. As you can see, AND operates like a house with a double-lock door. If neither lock is engaged (1 = unlocked), you can enter the house, but if either or both locks are engaged (0 = locked), you are shut outside.

OR sets the result bit to 1 for each position in which either or both operands contain a 1. Conversely, for bit positions where both operands contain 0, OR clears the result bit to 0.

Using the preceding operand combination,

```
00110100B OR 11010111B
```

produces 11110111B. To make another door analogy, OR operates like a house with two doors. If either or both doors are unlocked (1), you can enter the house, but if both are locked (0), you are shut outside.

XOR is a variation of OR in which 1's in both operands produce a 0 (rather than a 1) in the result. XOR is short for "exclusive-OR", because it excludes the 1-and-1 combination (to distinguish it from the "inclusive-OR" condition, OR.)

NOT reverses the state of each bit in the operand. That is, it changes each 1 to 0 and each 0 to 1. For example,

```
NOT 01101001B
```

produces 10010110B.

The 8088 also has assembly language instructions named AND, OR, XOR, and NOT, which we discuss in Chapter 3. The difference is that the logical operators do their job when the program *assembles*, whereas the logical instructions do theirs when the program executes.

Relational Operators

Relational operators compare two numeric values or memory addresses in the same segment, and produce a numeric result. The result is always one of two numbers: 0 if the relationship is "false" or 0FFFFH if the relationship is "true."

For example, if CHOICE is a predefined constant,

```
MOV  AX,CHOICE LT 20
```

assembles as

```
MOV  AX,0FFFFH
```

if CHOICE is less than 20, and as

```
MOV  AX,0
```

if CHOICE is greater than or equal to 20.

Because the relational operators can only produce two values (0 or 0FFFFH), they are rarely used alone. Instead, they are usually combined with other operators to form a decision-making expression. For instance, suppose you want AX to be loaded with 5 if CHOICE is less than 20 and with 6 otherwise. A statement that performs this task is:

```
MOV  AX,((CHOICE LT 20) AND 5) OR ((CHOICE GE 20) AND 6)
```

Here, if CHOICE is less than 20, the clause (CHOICE LT 20) is "true" and the clause (CHOICE GE 20) is "false," so the intermediate form of the statement is

```
MOV  AX,(0FFFFH AND 5) OR (0 AND 6)
```

The assembler evaluates this as

```
MOV AX,5
```

Conversely, if CHOICE is greater than or equal to 20, the clause (CHOICE LT 20) is "false" and (CHOICE GE 20) is "true," so the intermediate form of the statement is

```
MOV  AX,(0 AND 5) OR (0FFFFH AND 6)
```

The assembler evaluates this as

```
MOV AX,6
```

Value-Returning Operators

Operators in this group provide information about variables or labels in a program.

The dollar sign ($) returns the value of the location counter; that is, it returns the offset of the current statement. This operator is handy for making the assembler calculate the length of a text string. For example, the following calculates the number of characters in the MESSAGE string and assigns this count to MESSAGEL:

```
MESSAGE  DB  'Press any key to continue.'
MESSAGEL  EQU  $-MESSAGE
```

When we transmit the message to the screen, we can use the value MESSAGEL to tell our program how many characters to send.

SEG and *OFFSET* return the segment and offset values of a variable or label. For example, the statements

```
MOV  AX,SEG TABLE
MOV  BX,OFFSET TABLE
```

load the segment and offset values of TABLE into AX and BX, respectively. Of course, since both segment and offset are 16-bit values, they must be loaded into 16-bit registers.

The *TYPE* operator returns a numeric value that indicates the type attribute of a variable or the distance attribute of a label. For a variable, TYPE returns 1 (BYTE variable) or 2 (WORD). For a label, TYPE returns -1 (NEAR) or -2 (FAR).

The *LENGTH* and *SIZE* operators are only meaningful for variables you define with DUP. LENGTH returns the number of units (bytes or words) allocated for the variable. For example, the sequence

```
TABLE   DW    100 DUP(1)
        MOV   CX,LENGTH TABLE  ;Get no. of words in TABLE
```

loads 100 into CX. If you use LENGTH with any variable that was not defined with DUP, it returns 1.

SIZE returns the byte count of a variable. That is, it returns the product of LENGTH times TYPE. Using the variable TABLE we just defined, the statement

```
MOV   CX,SIZE TABLE  ;Get no. of bytes in TABLE
```

loads the value 200 into CX.

Attribute Operators

The pointer (PTR) operator overrides the type (BYTE or WORD) or distance (NEAR or FAR) attribute of an operand. For example, you can use PTR to refer to bytes in a table of words. If you define a table as

```
WORD_TABLE   DW   100 DUP(?)
```

the statement

```
FIRST_BYTE   EQU   BYTE PTR WORD_TABLE
```

assigns a name to the location of the first byte in WORD_TABLE. After that, you can name any other byte as easily as this:

```
FIFTH_BYTE   EQU   FIRST_BYTE+4
```

As we just mentioned, PTR can also change a label's distance attribute. For example, if we have this instruction in a program:

```
START:   MOV   CX,100
```

the label START has a NEAR attribute, which allows JMP (Jump) instructions in the same segment to refer to it. To let JMPs in other segments refer to this instruction as well, you must give it an alternate label that has a FAR attribute. This kind of statement will do the job:

```
FAR_START   EQU   FAR PTR START
```

As we mentioned in Chapter 1, when calculating a memory address, the 8088 automatically assumes that SS is the segment register if the operand employs SP or BP to hold the offset. Similarly, it assumes that DS is the segment register if BX, SI, or DI holds the offset. The *segment override* operator (DS:, ES:, SS:, or CS:) overrides the segment attribute of a label, variable, or address expression. It lets you specify an alternate segment register. For example,

```
MOV   AX,ES:[BP]
```

tells the 8088 to get its destination operand from the extra segment rather than from the stack segment.

The *SHORT* operator tells the assembler that a JMP target is within +127 or -128 bytes of the next instruction. With this information, the assembler encodes JMP as a two-byte instruction, rather than a three-byte instruction, which saves memory. Here is an example:

```
        JMP   SHORT THERE
        ..
        ..
THERE:
```

The *THIS* operator creates a memory address operand of a specified type (BYTE or WORD) or distance (NEAR or FAR), and gives this operand the same segment and offset attributes as the next memory address available for allocation. For example, the sequence

```
FIRST_BYTE   EQU   THIS BYTE
WORD_TABLE   DW    100 DUP(?)
```

creates FIRST_BYTE, and gives it a BYTE-type attribute with the same address as WORD_TABLE. This does the same thing as the previous statement

```
FIRST_BYTE   EQU   BYTE PTR WORD_TABLE
```

You can also use THIS to define FAR instruction locations. Returning to an earlier example,

```
START   EQU   THIS FAR
        MOV   CX,100
```

gives the MOV instruction a FAR attribute, which allows JMP instructions in another segment to jump to START directly.

The *HIGH* and *LOW* return the high- or low-order byte of a 16-bit expression. For example, if you define a constant as

```
CONST   EQU   0ABCDH
```

the statement

```
MOV   AH,HIGH CONST
```

loads the value 0ABH into the AH register.

2.7 Entering, Assembling, and Running a Program

Since we haven't discussed the details of the 8088's assembly language instruction set (that's coming in Chapter 3), you cannot yet write programs that add or subtract numbers, manipulate the registers, or perform the many other tasks you eventually want your computer to do. Still, you *do* have enough background to write a program that moves data with MOV instructions, and sets up the necessary segments with pseudo-ops.

In this section we will develop a program that copies one four-byte data table in memory into another. To make it more interesting, we will store the data into the "destination" table in the reverse order from how it is stored in the "source" table.

At any rate, the details of the program are unimportant. The main point is that you will learn how to enter a program into the computer, how to assemble it, how to produce a listing file and a run module, and how to execute and debug it. In short, you will get "hands-on" experience with the basic steps. That will help you to proceed with confidence through the more complex material found in the rest of the book.

Creating a Working Assembler Disk

For assembly language work, we need several programs that are on the IBM DOS disks (DOS 1.1 comes on one disk) as well as those on the Macro Assembler disk. If you have a hard disk, simply copy the entire contents of the Macro Assembler disk into whatever directory contains DOS. Otherwise, if you have floppy disks, you should prepare a disk that contains all the necessary programs. (Do this only if your computer has double-sided disk drives; a single-sided disk cannot hold all of the necessary program files.)

To begin, use the DOS *FORMAT /S* command to format a blank disk. Onto it, copy EDLIN.COM from the main DOS disk and DEBUG.COM

from the Supplemental Programs disk. What you do next depends on which version of the Macro Assembler you have:

- *If you have Version 1,* copy LINK.EXE from the Supplemental Programs disk, then copy MASM.EXE and CREF.EXE from the Macro Assembler disk.
- *If you have Version 2,* copy MASM.EXE, CREF.EXE, SALUT.EXE, and LIB.EXE from the Macro Assembler disk. Then copy LINK.EXE from the Supplemental Programs disk (if you have DOS 1.1) or from the Macro Assembler disk (if you have DOS 2.0 or higher).

From here on, we refer to the disk you just created as the *assembler disk.*

Data Disk

For the procedures in this section, you also need another blank disk that has been formatted with plain old *FORMAT* (no /S). This serves as your data disk.

Example Program

Figure 2-1 lists our table-copying program. Note that it has a stack segment (STACK), a data segment (DSEG), and a code segment (CSEG). The stack segment will hold a return address that makes the 8088 return to DEBUG upon completing the program.

The data segment consists of just two pseudo-op statements. One sets up the source table (SOURCE), the other reserves space for the destination table (DEST).

The code segment has four groups of instructions. They are:

- The first group puts DEBUG's return address on the stack.
- The second group makes DS point to the data segment. *(Remember, ASSUME doesn't do this for you.)*
- The third group stores zeros in the four bytes of the DEST table. We do this so that when you check the final contents of DEST, you know you're seeing the program's effect on DEST, not the results of a previous run.
- The fourth group copies the table data, one byte at a time.

```
TITLE   EX_PROG - Example Program
          PAGE      ,132
STACK     SEGMENT  PARA STACK 'STACK'
         DB  64 DUP('STACK  ')
STACK       ENDS
DSEG        SEGMENT  PARA PUBLIC 'DATA'
```

```
SOURCE  DB  10,20,30,40    ;This table will be copied into
DEST    DB  4 DUP(?)       ; this table, in reverse order
DSEG    ENDS
SUBTTL Here is the main program.
        PAGE
CSEG    SEGMENT  PARA PUBLIC 'CODE'
OUR_PROG PROC    FAR
        ASSUME   CS:CSEG,DS:DSEG,SS:STACK
;
;  Set up the stack to contain the proper values so this
;  program can return to DEBUG.
;
        PUSH  DS          ;Put return seg. addr. on stack
        MOV   AX,0        ;Clear a register
        PUSH  AX          ;Put zero return addr. on stack
;
;  Initialize the data segment address.
;
        MOV   AX,DSEG     ;Initialize DS
        MOV   DS,AX
;
;  Initialize DEST with zeroes.
;
        MOV   DEST,0      ;First byte
        MOV   DEST+1,0    ;Second byte
        MOV   DEST+2,0    ;Third byte
        MOV   DEST+3,0    ;Fourth byte
;
;  Copy SOURCE table into DEST table, in reverse order.
;
        MOV   AL,SOURCE     ;Copy first byte
        MOV   DEST+3,AL
        MOV   AL,SOURCE+1   ;Copy second byte
        MOV   DEST+2,AL
        MOV   AL,SOURCE+2   ;Copy third byte
        MOV   DEST+1,AL
        MOV   AL,SOURCE+3   ;Copy fourth byte
        MOV   DEST,AL
        RET                 ;Return to DEBUG
OUR_PROG ENDP
CSEG    ENDS
        END  OUR_PROG
```

Figure 2-1. Example program for editing and assembling.

The instructions in the code segment are a single procedure enclosed by the pseudo-op statements *OUR_PROG PROC FAR* and *OUR_PROG ENDP*. We gave the procedure a FAR attribute so that the RET instruction at the end will return the 8088 to DEBUG (which is in a different segment of memory) after the program has executed. *For this same reason, you should set up the programs you write as procedures.*

Entering the Program

To enter the program into the computer, you can use either EDLIN or, better yet, any editor or word processor that can produce standard ASCII text (unformatted text that contains no control characters). For example, you could use WordStar's "non-document" (N) mode. Since we can't know which word processor you have (if any), we will use EDLIN. Table 2-6 summarizes the most useful EDLIN commands.

Generally, we will also assume that you have two floppy disk drives. If you have only one, you can still use our procedures, but you will have to do a certain amount of disk-swapping. If you have a hard disk, ignore references to A> and B>, which are floppy disk prompts. Instead, mentally substitute C> wherever the book mentions A> or B>.

To enter the program, proceed as follows:

1. Insert your new assembler disk in the left-hand drive (A) and your blank data disk in the right-hand drive (B), then switch the power on. Press Return when the computer asks for the date and time.
2. When the A> prompt appears, type **b:** and press Return to make the right-hand drive active. (Hereafter, press Return after every command.)
3. Enter

   B>a:edlin ex_prog.asm

   Here, *ex_prog* is the name of the program we want to create. The extension *.asm* identifies it as an assembly language source code file.
4. When the computer displays

   ```
   New file
   *_
   ```

   press I to put EDLIN into the *insert* mode. The next prompt,

   ```
   1:*_
   ```

   indicates that EDLIN is waiting for the first line of "text"—in this case, the first statement in your source program.

Table 2-6. Common EDLIN commands.

Command	*Action*	*Comments*
[start-line][,end-line] D	Deletes lines.	D alone deletes current line.
[line]	Displays a line for editing.	Pressing Return selects the next line.
E	Saves the program on disk, then returns to DOS.	Use E to end an editing session. See also Q below.
[line] I	Inserts lines from the keyboard ahead of the specified line. Press Ctrl-Break to leave the insert mode.	Use I alone (no line number) to start a new source program.
[start-line][,end-line] L	Lists (diplays) lines.	
Q	Returns to DOS without saving editing changes.	See also **E** above.
[start-line],[end-line],target-line C	Copies the specified lines to a point just ahead of *target-line*.	DOS 2.0 and above.
[start-line],[end-line],target-line M	Moves the specified lines to a point just ahead of *target-line*.	DOS 2.0 and above.
[start-line][,end-line] R old-string *then* **<F6>new-string**	Replaces *old-string* with *new-string* throughout the specified range.	
[start-line][,end-line] S string	Searches for the specified string.	

Note: Brackets indicate that the item(s) inside them are optional.

5. Enter the source program shown in Figure 2-1, line by line. We have capitalized most words, but you may enter them in lowercase if you prefer. We have also aligned the fields to make the listing more readable, but feel free to enter them any way you like. Just be sure you put at least one space between them.
6. When you finish, press Ctrl-Break to take EDLIN out of the insert mode, then press **E** to save the program on the data disk.

Now you can assemble this *source program* to produce an *object program.*

Assembling the Program

To assemble the program, type

```
B>a:masm ex_prog
```

Then respond to three prompts, as follows:

```
Object filename [EX_PROG.OBJ]: (press Return)
Source listing  [NUL.LST]: ex_prog
Cross reference [NUL.CRF]: (press Return)
```

This tells the assembler to use the specified source file (EX_PROG.ASM) to create an object file with the suggested name (EX_PROG.OBJ) and a source listing file (EX_PROG.LST). The listing file contains the source instructions and their numeric codes—a handy, printable file to see how the assembler interpreted your program.

If you followed the preceding steps exactly, the assembler will do its job, then display

```
Warning Severe
Errors  Errors
0       0
```

then return to DOS. If it reports errors, use EDLIN to correct them, then assemble EX_PROG.ASM again.

Listing the Source Program

To list the program on the screen, enter

```
B>type ex_prog.lst
```

The instructions fly by so quickly you can't read them! To stop the scrolling temporarily, press Ctrl-Num Lock; press any key to resume. To print the listing, first set your printer "on-line," then press Ctrl-PrtSc.

The listing should look like Figure 2-2. Note that a PAGE pseudo-op in the source program makes it three pages long instead of two.

The first two pages list the source program instructions and their associated object code, in this order:

- The leftmost column shows the hexadecimal offset values (in bytes) from the beginning of the segment.
- The numeric columns show the object code for each statement. For the stack and data segments, these are the values in each memory location. For the code segment, the numbers represent the machine language codes that the 8088 will execute.
- The text to the right is, of course, the source program.

The third page of the listing gives detailed summary information about the segments and symbols in the program. You can generally ignore this part of the listing. In fact, Version 2 of the assembler provides an option (/N) that omits these tables entirely.

Finding the Program's Stopping Point

The main thing we want from this listing is the offset of the RET instruction at the end of the program. We will use this value later to execute the program until it reaches RET. There we will examine the contents of registers and the data tables. We must stop before executing RET because RET puts the 8088 back into the system program, at which point we won't be able to find these values easily.

Use a Type command to display the listings. When the RET instruction appears, note that its offset (the leftmost number) is *0036*. Keep this number in mind for later use.

Creating the Run File

DOS can store an object program at any convenient place in memory. To use this facility, you must create a *relocatable* run file.

The program that creates relocatable run files is called the *Linker*, because it can link several object files into one big run file. To run the Linker, enter

```
B>a:link ex_prog;
```

When B> appears, your data disk contains the run file *EX_PROG.EXE*.

```
 IBM Personal Computer MACRO Assembler   Version 2.00    Page    1-1
EX_PROG - Example Program                                12-17-84


                                TITLE  EX_PROG - Example Program
                                   PAGE      ,132
0000                            STACK     SEGMENT   PARA STACK 'STACK'
0000     40 [                             DB        64 DUP('STACK   ')
              53 54 41 43
              4B 20 20 20
                           ]

0200                            STACK     ENDS
0000                            DSEG      SEGMENT   PARA PUBLIC 'DATA'
0000  0A 14 1E 28               SOURCE    DB        10,20,30,40      ;This table will be copied into
0004     04 [                   DEST      DB        4 DUP(?)         ; this table, in reverse order
                 ??
                     ]

0008                            DSEG      ENDS
                                SUBTTL  Here is the main program.
```

Figure 2-2. Assembly listing for the example program (page 1 of 3).

```
 IBM Personal Computer MACRO Assembler   Version 2.00    Page    1-2
EX_PROG - Example Program                                12-17-84
                        Here is the main program.

                                PAGE
0000                    CSEG    SEGMENT   PARA PUBLIC 'CODE'
0000                    OUR_PROG PROC     FAR
                                ASSUME    CS:CSEG,DS:DSEG,SS:STACK
                        ;
                        ;  Set up the stack to contain the proper values so this
                        ;  program can return to DEBUG.
                        ;
0000  1E                        PUSH      DS              ;Put return seg. addr. on stack
0001  B8 0000                  MOV       AX,0            ;Clear a regster
0004  50                        PUSH      AX              ;Put zero return addr. on stack
                        ;
                        ;  Initialize the data segment address.
                        ;
0005  B8 ---- R                 MOV       AX,DSEG         ;Initialize DS
0008  8E D8                     MOV       DS,AX
                        ;
                        ;  Initialize DEST with zeroes.
                        ;
000A  C6 06 0004 R 00           MOV       DEST,0          ;First byte
000F  C6 06 0005 R 00           MOV       DEST+1,0        ;Second byte
0014  C6 06 0006 R 00           MOV       DEST+2,0        ;Third byte
0019  C6 06 0007 R 00           MOV       DEST+3,0        ;Fourth byte
                        ;
                        ;  Copy SOURCE table into DEST table, in reverse order.
                        ;
001E  A0 0000 R                 MOV       AL,SOURCE       ;Copy first byte
0021  A2 0007 R                 MOV       DEST+3,AL
0024  A0 0001 R                 MOV       AL,SOURCE+1     ;Copy second byte
0027  A2 0006 R                 MOV       DEST+2,AL
002A  A0 0002 R                 MOV       AL,SOURCE+2     ;Copy third byte
002D  A2 0005 R                 MOV       DEST+1,AL
0030  A0 0003 R                 MOV       AL,SOURCE+3     ;Copy fourth byte
0033  A2 0004 R                 MOV       DEST,AL
0036  CB                       RET                       ;Far return to DEBUG
0037                    OUR_PROG ENDP
0037                    CSEG    ENDS
                                END       OUR_PROG
```

Figure 2-2. Assembly listing for the example program (page 2 of 3).

```
 IBM Personal Computer MACRO Assembler    Version 2.00    Page    Symbols-1
EX_PROG - Example Program                                       12-17-84


Segments and Groups:

                N a m e                 Size    Align   Combine Class

CSEG . . . . . . . . . . . . . .        0037    PARA    PUBLIC  'CODE'
DSEG . . . . . . . . . . . . . .        0008    PARA    PUBLIC  'DATA'
STACK. . . . . . . . . . . . . .        0200    PARA    STACK   'STACK'


Symbols:

                N a m e                 Type    Value   Attr

DEST . . . . . . . . . . . . . .        L BYTE  0004    DSEG    Length =0004
OUR_PROG . . . . . . . . . . . .        F PROC  0000    CSEG    Length =0037
SOURCE . . . . . . . . . . . . .        L BYTE  0000    DSEG

50092 Bytes free

Warning Severe
Errors  Errors
0       0

A>
```

Figure 2-2. Assembly listing for the example program (page 3 of 3).

Linking Multiple Object Modules

This particular example has only one object module (EX_PROG.OBJ), but in later chapters you will create programs that have two or more object modules. You assemble these modules separately, then link them by listing their names connected with + symbols. For example,

```
B>a:link mod1+mod2+mod3;
```

links object modules MOD1.OBJ, MOD2.OBJ, and MOD3.OBJ to create a run file called MOD1.EXE.

Running the Program

As we mentioned earlier, you may run your relocatable run file (EX_PROG.EXE) under DOS or DEBUG. You generally run a program under DOS only when you are sure that it is free of errors or it produces some visible result. The example program is error-free (knock on formica), but it returns results in memory, where they are not visible. Hence, we will run it under DEBUG. Table 2-7 summarizes the most common DEBUG commands.

To begin, enter

```
a:debug ex_prog.exe
```

The computer shows DEBUG's hyphen (-) prompt.

Figure 2-3 shows a typical DEBUG session. Let's look at what's happening step by step. Don't pay much attention to the addresses, however; they may be different on your computer.

To begin, type **R** to display the registers. DEBUG's R (Registers) command shows the contents of each register as four hexadecimal digits. It also summarizes the bit settings in the Flags register as a set of two-letter abbreviations. Table 2-8 tells what these abbreviations mean. For example, the abbreviations produced by this Register command tell you that all Flags bits are initially in the "Clear" (0) state.

The final line of the Register command display shows which instruction the 8088 will execute next (*not* the instruction it just executed). Reading from left to right, you see that the next instruction is at address 0923:0000 (these are the CS and IP values, respectively), its object code is 1E, and it is a PUSH DS. Things must be working fine so far, because PUSH DS is indeed the first instruction in our example program.

Table 2-7. Common DEBUG commands.

Command	*Action*	*Comments*
D address *or* **D start-address end-offset**	Displays the contents of memory locations.	Note that the end address is an offset only.
E address value1 [values2...]	Changes the contents of memory location(s).	
F start-address L bytes value	Fills a block of memory with a specified value.	
G [offset1] [offset2...]	Executes a program. Offset values are *breakpoints*. If you include them, the processor stops before executing that instruction and displays registers.	See **T** below.
Q	Exits DEBUG and returns to DOS.	
R [register-name]	Displays contents of one or all registers.	If you display just one register, R allows you to change its contents.
T [instruction-count]	Executes one or more instructions and displays register values with each.	See **G** above.
U [address]	Translates memory contents into instructions (i.e., unassembles memory).	

Note: Brackets indicate that the item(s) inside them are optional.

Table 2-8. Flags abbreviations used by DEBUG.

Flag Name	*Set*	*Clear*
Overflow (yes/no)	OV	NV
Direction (decrement/increment)	DN	UP
Interrupt (enable/disable)	EI	DI
Sign (negative/positive)	NG	PL
Zero (yes/no)	ZR	NZ
Auxiliary Carry (yes/no)	AC	NA
Parity (even/odd)	PE	PO
Carry (yes/no)	CY	NC

```
  debug ex_prog.exe
-r
AX=0000  BX=0000  CX=0250  DX=0000  SP=0200  BP=0000  SI=0000  DI=0000
DS=0913  ES=0913  SS=0928  CS=0923  IP=0000   NV UP DI PL NZ NA PO NC
0923:0000 1E            PUSH    DS
-t

AX=0000  BX=0000  CX=0250  DX=0000  SP=01FE  BP=0000  SI=0000  DI=0000
DS=0913  ES=0913  SS=0928  CS=0923  IP=0001   NV UP DI PL NZ NA PO NC
0923:0001 B80000        MOV     AX,0000
-t

AX=0000  BX=0000  CX=0250  DX=0000  SP=01FE  BP=0000  SI=0000  DI=0000
DS=0913  ES=0913  SS=0928  CS=0923  IP=0004   NV UP DI PL NZ NA PO NC
0923:0004 50            PUSH    AX
-g36

AX=0928  BX=0000  CX=0250  DX=0000  SP=01FC  BP=0000  SI=0000  DI=0000
DS=0927  ES=0913  SS=0928  CS=0923  IP=0036   NV UP DI PL NZ NA PO NC
0923:0036 CB            RETF
-d ds:0
0927:0000  0A 14 1E 28 28 1E 14 0A-00 00 00 00 00 00 00 00   ...((..........
0927:0010  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
0927:0020  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
0927:0030  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
0927:0040  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
0927:0050  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
0927:0060  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
0927:0070  53 54 41 43 4B 20 20 20-53 54 41 43 4B 20 20 20   STACK   STACK
-q

A>
```

Figure 2-3. A DEBUG session with the example program.

Next we use T (Trace) commands to execute the *PUSH DS* and the *MOV AX,0000* instruction that follows it. Each time you enter T, the computer executes one instruction, then it displays the registers and lists the instruction it will execute next. Thus, tracing lets you see what the registers contain at any given time, and which path the processor is taking. This information is invaluable in debugging a program.

Notice how the registers change as each instruction executes. After the first T, the Stack Pointer (SP) has decreased from 0200 to 01FE as a result of the PUSH operation. If you traced through the PUSH AX instruction, you would see SP decrease again.

After tracing through these first two instructions, we use the G (Go) command to execute down to the RET instruction. Specifically, we enter *g36*, because, as we noted earlier, RET has an offset of 36 from the beginning of the code segment. DEBUG shows RET as *RETF*, because it is making a return from a FAR procedure.

With the program completed, we must check to see if it worked correctly—that is, whether it copied the SOURCE table to the DEST table in reverse order. To do this, we use the D (Dump) command to view the final contents of the tables. Enter

d ds:0

This tells DEBUG to display (d) the contents of the data segment (ds), starting at the beginning (offset 0).

The top line of the display shows

```
0A 14 1E 28 28 1E 14 0A
```

The first four values are the contents of SOURCE; the next four are the contents of DEST. These hexadecimal numbers represent the decimal values

```
10 20 30 40 40 30 20 10
```

As you can see, the program did its job.

Besides displaying numeric values for memory locations, the Dump command also shows their character equivalents, if any. This particular display is showing the unused portion at the bottom of the stack segment, because the right-hand side of the dump shows repetitions of the word STACK. We made the stack display itself this way by defining the stack segment with the pseudo-op

```
DB 64 DUP('STACK   ')
```

Finally, we use the Q (Quit) command to leave DEBUG and return to DOS.

Advanced Listing Options

If your program is long or complex, you can produce two other listings that provide additional information. These are the *cross reference listing* and the *map listing*.

Cross-Reference Listing

The cross-reference listing tells the line number where each symbol is defined and any other lines that refer to it. To generate a cross reference listing, you have to produce a cross-reference (.CRF) file. To do this, enter the program name in response to the assembler's "Cross-reference" prompt. For our example program, the correct response would be

```
Cross reference [NUL.CRF]: ex_prog
```

Besides creating the CRF file, entering a name in response to this prompt makes the assembler put line numbers in its regular assembly (.LST) listing.

Then, after assembling, type

```
B>a:cref
```

and respond to the prompts as follows:

```
Cref filename [.CRF]: ex_prog
List filename [EX_PROG.REF]: (Press Return)
```

When the DOS prompt reappears, display the cross-reference listing by entering

```
type ex_prog.ref
```

As Figure 2-4 shows, you get the name of each symbol in the program, along with the line number where it is defined (marked with a #) and the line numbers of any other statements that refer to it.

Map Listing

The map listing summarizes the program's segments. For each segment, it shows the "start" and "stop" offsets, the length of the segment in bytes, and its class (CODE, DATA, EXTRA, or STACK).

The map listing file is a LINK option. You get it by entering the form

```
link progname,,;
```

(Note the two extra commas preceding the semicolon.)

To display the map listing, enter

```
type ex_prog.map
```

If you want the listing printed, first press Ctrl and PrtSc.

```
 EX_PROG - Example Program

 Symbol Cross Reference                          (# is definition)       Cref-1

CODE . . . . . .  19
CSEG . . . . . .  19#     21      54

DATA . . . . . .  10
DEST . . . . . .  12#     37      38      39      40      45      47      49      51
DSEG . . . . . .  10#     16      21      32

OUR_PROG . . . . . .  20#     53      55

SOURCE . . . . . .  11#     44      46      48      50
STACK. . . . . .   3#      3       9      21

8 Symbols

62578 Bytes Free

A>
```

Figure 2-4. Cross-reference listing for the example program.

The listing should look like Figure 2-5. When it is done printing, press Ctrl and PrtSc again to make DOS stop sending screen text to the printer.

```
type ex_prog.map


 Start  Stop   Length  Name                    Class
 00000H 00036H 0037H   CSEG                    CODE
 00040H 00047H 0008H   DSEG                    DATA
 00050H 0024FH 0200H   STACK                   STACK

 Origin   Group

Program entry point at 0000:0000

A>
```

Figure 2-5. Map listing for the example program.

2.8 Models for Constructing Programs

We will now present generalized models that you can use to enter programs from this book or those you design. The models include the "boilerplate" that every program should have. All you have to do is fill in the data and instructions that apply to your particular program.

Example 2-1 shows the model for a source module that will contain a complete program, or that will be linked with one or more secondary modules to form a program. Note that if this main module contains references to items in a secondary module, it must include an EXTRN statement that lists them. On the optional disk, this model is named *MAINMOD.ASM*.

Example 2-2 shows the model for a secondary module that must be linked with the main module in Example 2-1 to form a program. Note the following:

1. Because this module's code segment has the same name (CSEG) as the one in the main module, we have declared the PNAME procedure NEAR. To call it from a code segment that has a different name, change PNAME to FAR.
2. Because this module's data segment has the same name (DSEG) as the one in the main module, we did not initialize DS. There was no need to; instructions in the main module initialize it. However, if you use a different data segment name in your main module, insert DS-loading instructions in this secondary module.
3. We have represented the procedure name with the word PNAME. When you create your own module, replace PNAME with the name

you want in three places: the PUBLIC statement at the top, the PROC statement in the middle, and the ENDP statement at the end.
4. This module has a PUBLIC statement that corresponds to the main module's EXTRN. If the main module contains a call to PNAME, it must include the statement EXTRN PNAME:NEAR.
5. This module's END pseudo-op is unlabeled, because it is not the main module.

On the optional disk, this model is named *SECMOD.ASM*.

You can create these models as you would any other program: using EDLIN, an editor, or a word processing program. Then, when you want to enter a program, copy the appropriate model and give the copy the name you want (e.g., **copy mainmod.asm newprog.asm)**. Finally, use your editor to insert your instructions and data.

Example 2-1. Model for main program module.

```
TITLE (Insert title here.)
                PAGE            ,132
(Insert EXTRN statement, if appropriate.)

STACK           SEGMENT   PARA STACK 'STACK'
                DB        64 DUP('STACK   ') ;Stack area
STACK           ENDS

DSEG            SEGMENT   PARA PUBLIC 'DATA'

(Insert data here.)

DSEG            ENDS

CSEG            SEGMENT   PARA PUBLIC 'CODE'
                ASSUME    CS:CSEG,DS:DSEG,SS:STACK

ENTRY           PROC      FAR           ;Entry point
; Set up the stack to contain the proper values so this
; program can return to DOS or DEBUG.

                PUSH      DS
                SUB       AX,AX
                PUSH      AX

; Initialize the data segment address.

                MOV       AX,DSEG
```

```
        MOV     DS,AX

(Insert instructions here.)

        RET                     ;Return to DOS or DEBUG
ENTRY   ENDP
CSEG    ENDS
        END     ENTRY
```

Example 2-2. Model for secondary module.

```
TITLE   (Insert title here.)
        PAGE    ,132
        PUBLIC  PNAME
(Insert PUBLIC for data segment variables, if appropriate.)

DSEG    SEGMENT PARA PUBLIC 'DATA'

(Insert data here.)
DSEG    ENDS

CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CSEG,DS:DSEG

PNAME   PROC    NEAR

(Insert instructions here.)

        RET                     ;Return to calling program
PNAME   ENDP
CSEG    ENDS
        END
```

2.9 Advanced Pseudo-Ops

In Section 2.5 we discussed the assembler pseudo-ops that you will probably use most often. This section describes additional pseudo-ops that are less common, or used by advanced programmers. Table 2-9 lists these advanced pseudo-ops in three groups: data, conditional, and listing.

Table 2-9. Advanced pseudo-ops.

Type	*Pseudo-ops*		
Data	EVEN	LABEL	
	GROUP	ORG	
Conditional	ELSE	IFNDEF	IF1
	ENDIF	IFDIF	IF2
	IF	IFE	
	IFDEF	IFIDN	
Listing	.CREF	%OUT	.XLIST
	.LFCOND	.SFCOND	
	.LIST	.XREF	

Data Pseudo-Ops

We can divide the assembler's advanced data pseudo-ops into *Block Definition* and *Assembly Control,* as shown in Table 2-10.

Table 2-10. Advanced data pseudo-ops.

Pseudo-Op	*Function*
Block Specification	
GROUP	Format: `name GROUP seg-name[,...]` Collects the specified segments under one name, so they all reside within a 64K-byte physical segment.
LABEL	Format: `name LABEL type` Defines the attributes of *name.*
Assembly Control	
ORG	Format: `ORG expression` Sets location counter to the value of *expression.* assembler will store subsequent object code starting at that address.
EVEN	Format: **EVEN** Forces location counter to an even boundary.

Block Specification Pseudo-Ops

The GROUP pseudo-op collects segments under one name so they are all in a single 64K-byte block of memory. You will need GROUP if you want to

develop a program as a DOS *COM* (Command) file, which requires the program to be in one block.

For example, if your program has a code segment named CODESEG and a data segment named DATASEG, you can group them into a 64K block called CGROUP as follows:

```
CGROUP   GROUP    CODESEG, DATASEG
DATASEG  SEGMENT  PARA PUBLIC 'DATA'
         ..
         ..
DATASEG  ENDS
CODESEG  SEGMENT  PARA PUBLIC 'CODE'
         ASSUME   CS:CGROUP, DS:CGROUP
         ..
         ..
CODESEG  ENDS
         END
```

The LABEL pseudo-op defines the *segment*, *offset*, and *type* attributes of *name*. You can use LABEL to give an instruction a FAR attribute, so that a jump instruction in another segment can transfer to it. For example,

```
HERE  LABEL  FAR
      MOV    DX,0
```

labels the MOV instruction *HERE*.

You can also use LABEL to access bytes in a table of word values, or vice versa. For example, the 8088 will recognize the following as either a byte table called B_TABLE or a word table called W_TABLE:

```
B_TABLE  LABEL  BYTE
W_TABLE  DW     2F24H,36AH,0817H,3
```

Assembly Control Pseudo-Ops

The ORG (Origin) pseudo-op alters the *location counter*, the internal pointer that tells the assembler where to store instructions and data in memory. Normally you let DOS make the placement decisions, but ORG gives you the option of making these decisions yourself. For example, if you want to use your program as a DOS *COM* file (see the DOS manual, especially the EXE2BIN command), precede the first instruction with

```
ORG  100H
```

This tells the assembler to start storing the program 256 bytes past the beginning of the code segment.

EVEN is a seldom-used pseudo-op you can use to make programs run more efficiently on an *8086*-based computer, or on an *80286*-based computer such as the IBM PC AT. Having a 16-bit data bus, the 8086 can transfer 16 bits of data in one operation (as opposed to the 8088, which requires two 8-bit operations to transfer 16 bits). However, the 8086 takes longer to transfer data that starts at an odd-numbered address than data that starts at an even-numbered address. Therefore, in time-critical applications it is advantageous to store data at even-numbered addresses.

To keep data properly aligned, arrange your data segment with doublewords first, words second, and bytes last. If the segment contains only byte pseudo-ops, put an EVEN before each one except the first. For example:

```
DSEG  SEGMENT  PARA PUBLIC 'DATA'
      HOURS  DB  ?
      EVEN
      MESSAGE  DB  'Press any key to continue.'
DSEG  ENDS
```

If the location counter is already pointing to an even offset, EVEN does nothing, but if it is pointing to an odd offset, the assembler replaces EVEN with a 00H byte to make the next location even. For example, if the location counter is pointing to offset 129H, EVEN makes the assembler store the next data unit at 12AH.

Conditional Pseudo-Ops

Conditional pseudo-ops cause the assembler to either assemble or skip a set of source statements, based on whether a specified condition is "true" or "false" at assembly time. This selective "assemble/don't-assemble capability" lets you place diagnostics or special conditions in test runs, or create specialized versions of a multipurpose program.

To "conditionally assemble" a portion of a program, precede it with an *IF* pseudo-op (see Table 2-11) and follow it with an *ENDIF* pseudo-op. In each case, if the test condition evaluates as "true," the enclosed code will be assembled; if the condition evaluates as "false," the assembler will skip the code and continue with the statement that follows ENDIF.

We can group the eight IF pseudo-ops into four pairs, as follows:

- IFE is true if *expression* is 0; IF is true if *expression* is not 0.
- IF1 is true when the assembler is executing pass 1; IF2 is true when it is executing pass 2.

- IFDEF is true if *symbol* is defined or has been declared external by an EXTRN pseudo-op; otherwise, IFNDEF is true.
- IFIDN is true if strings *argument-1* and *argument-2* are identical; IFIDF is true if they are different.

For example, to include diagnostic routines in a test run, enclose them with IFE and ENDIF, and set up a constant called FOR_TEST_ONLY. At assembly time, the assembler checks the value of FOR_TEST_ONLY. If it is zero, the diagnostics are included in (assembled into) the program. If FOR_TEST_ONLY has any other value, the diagnostics are omitted. The program might have this form:

```
        IFE FOR_TEST_ONLY
DIAG1:  ..    (diagnostic test instructions)
        ..
        ENDIF
```

The instructions between DIAG1 and ENDIF are assembled only if a statement such as

```
FOR_TEST_ONLY   =   0
```

appears earlier in the program. A statement such as

```
FOR_TEST_ONLY   =   1
```

makes the assembler skip the instructions between DIAG1 and ENDIF.

ELSE Option

You can include an ELSE term to generate an *alternate* set of instructions for the "false" case. The general format is:

```
IFxx    [argument]
   ..   (Statements for "true" case)
   ..
[ELSE]
   ..   (Statements for "false" case)
   ..
ENDIF
```

The ELSE option allows you, for example, to produce two versions of a program: one that displays prompts and messages in English and another

Table 2-11. Conditional pseudo-ops.

Pseudo-Op	*Function*
IFE	Format: `IFE expression` True if *expression* is zero.
IF	Format: `IF expression` True if *expression* is not zero.
IF1	Format: `IF1` True if assembler is executing pass one.
IF2	Format: `IF2` True if assembler is executing pass two.
IFDEF	Format: `IFDEF symbol` True if *symbol* is defined or has been declared external by an EXTRN pseudo-op.
IFNDEF	Format: `IFNDEF symbol` True if *symbol* is undefined or not declared external by an EXTRN pseudo-op.
IFIDN	Format: `IFIDN <string1>,<string2>` True if *string1* and *string2* are identical. Angle brackets are required.
IFIDF	Format: `IFIDF <string1>,<string2>` True if *string1* and *string2* are different. Angle brackets are required.

that displays them in Spanish. To do this, set up a constant called LANGUAGE to select the instructions that deal with each language. If LANGUAGE is zero, the assembler produces the English version; if LANGUAGE is one, it produces the Spanish version. A message-related section of the program may have this form:

```
IFE  LANGUAGE
   ..    (English-producing  statements)
   ..
ELSE
   ..    (Spanish-producing  statements)
   ..
ENDIF
```

Nesting Conditionals

You can give the assembler more than two options by *nesting* conditional clauses. For example, suppose the Spanish version of your program really

caught on, and you decide to produce other versions that display messages and prompts in French and German. To do this, modify the program so the assembler selects the language based on whether LANGUAGE has a value of 0, 1, 2, or 3 (for English, Spanish, French, or German). Now the message section may have this form:

```
IFE LANGUAGE
  ..      (English-producing statements)
  ..
ELSE
  IFE LANGUAGE-1
    ..      (Spanish-producing statements)
    ..
  ELSE
    IFE LANGUAGE-2
      ..    (French-producing statements)
      ..
    ELSE
      ..    (German-producing statements)
      ..
    ENDIF
  ENDIF
ENDIF
```

Note that we need three separate ENDIFs to balance the preceding IFEs. Note also that we have to indent the IFE-ENDIF combinations. The assembler does not require the indentation; we just did it to make the program easier to read.

Listing Pseudo-Ops

Listing pseudo-ops tell the assembler what to print and how to format it. They are summarized in Table 2-12 as four functional groups.

Listing Control Pseudo-Ops

The pseudo-ops .XCREF and .CREF let you exclude portions of a program from the cross-reference file (.CREF), while .XLIST and .LIST let you exclude them from the assembler listing file (.LST). You can use .LIST and .XLIST to print selected procedures within a long program; put .LIST at the beginning of a procedure and .XLIST at the end.

Table 2-12. Listing pseudo-ops.

Pseudo-Op	*Function*
Listing Control	
.XCREF	Format: `.XRCEF` Turns off cross-reference listing between here and the next .CREF (if any).
.CREF	Format: `.CREF` Restores cross-reference listing.
.XLIST	Format: `.XLIST` Turns off the assembly listing between here and the next .LIST (if any).
.LIST	Format: `.LIST` Restores assembly listing.
Display Status Message	
%OUT	Format: `%OUT text` Displays the *text* message.
False Conditional Block Control	
.LFCOND	Format: `.LFCOND` Lists all conditional blocks. This is the default setting.
.SFCOND	Format: `.SFCOND` Omits "false" conditional blocks from the listing.

Display Status Message Pseudo-Op

The %OUT pseudo-op displays a message while a program is being assembled. This is useful for reporting on the progress of a long assembly. For example, the following tells you when the assembler has reached the halfway point in its two-pass assembly procedure.

```
IF2
  %OUT  Starting assembly pass 2
ENDIF
```

If you see the message, you wait; otherwise, take a walk out to the coffee machine.

False Conditional Block Control Pseudo-Ops

From our preceding discussion of conditional assembly, you know that if an IF pseudo-op evaluates as "false," the assembler ignores everything between IF and the next ENDIF. However, you will probably want to *list* those unassembled statements eventually, so you can see your program in its entirety.

.LFCOND and .SFCOND control the listing of conditional blocks. Quite simply, .LFCOND causes everything in conditional blocks to be listed, while .SFCOND omits false conditional blocks from the listing.

2.10 Key Point Summary

Following are the key points you learned in this chapter:

1. Lines in a program, or *statements,* can represent either assembly language instructions or pseudo-ops. Assembly language instructions are commands to the computer's microprocessor, while pseudo-ops are commands to the assembler.
2. Each assembly language statement can have up to four fields, as follows:

   ```
   [Label:] Mnemonic [Operand] [;Comment]
   ```

   The label field assigns a name to an instruction, which lets other instructions refer to it. Every instruction label must be followed with a colon. The mnemonic field contains the two- to six-letter acronym for the instruction. The operand field tells the 8088 where to find the data it is to operate on. The comment field lets you include a short description of the instruction; it must be preceded by a semicolon.
3. Each pseudo-op statement can have up to four fields, as follows:

   ```
   [Name] Pseudo-op [Operand] [;Comment]
   ```

4. The most frequently-used pseudo-ops can be divided into two groups: data and listing.
5. *Data pseudo-ops* can be divided into five groups: symbol definition, data definition, external reference, segment/procedure specification, and assembly control.
6. There are two *symbol definition pseudo-ops,* EQU and =. EQU assigns a name to an expression permanently, while = assigns one temporarily; you can redefine it later.
7. There are three *data definition pseudo-ops*: DB (Define Byte), DW (Define Word), and DD (Define Doubleword). DB and DW are generally used

to allocate memory space for variables, while DD is used to allocate space for an address. In all three cases, you can either specify an initial value or simply reserve space in memory. To reserve space, put a question mark (?) in the operand field.

To set up a table, enter the elements with commas between them. To make the assembler repeat a value, tell it how many times to repeat using a DUP operator.

8. The *segment/procedure specification pseudo-ops* include SEGMENT and ENDS (used to define segments), PROC and ENDP (used to define procedures), and ASSUME (which identifies the kinds of segments (code, data, extra, or stack) you are using).

   Procedures can be NEAR or FAR. NEAR procedures can be called only from within the code segment, while FAR procedures can be called from other code segments.
9. *External reference pseudo-ops* allow you to use information (e.g., a procedure or a variable) that is in a file elsewhere in the system. PUBLIC makes symbols available to other modules that will eventually be linked to this one. Its opposite, EXTRN, specifies the external symbols a module refers to. INCLUDE merges an external file into the current source file at assembly time.
10. The *assembly control pseudo-op* END marks the end of a source module, so it must appear in every module.
11. The *listing pseudo-ops* control the format of assembly listings. PAGE specifies the length and width of the page, while TITLE and SUBTTL can be used to insert a title or subtitle.
12. There are five kinds of *operators:* arithmetic, logical, relational, value-returning, and attribute.
13. *Arithmetic operators* include the standard ones for addition, subtraction, multiplication, and division (+, -, *, and /), plus MOD, which returns the remainder of a division.
14. The *logical operators* AND, OR, XOR, and NOT let you manipulate individual bits in a binary number.
15. The *relational operators* compare the magnitude of two operands with tests for equal (EQ), not equal (NE), less than (LT), greater than (GT), less than or equal to (LE), or greater than or equal to (GE).
16. The two most common *value-returning operators* are SEG and OFFSET, which return the segment or offset value of a variable or label.
17. The most useful *attribute operators* are DS:, ES:, SS:, and CS:. They let you specify a segment that is different from the one the processor normally uses to transfer data to or from a memory location.
18. To develop a program, you must enter it into the computer (using EDLIN or a word processor), then assemble it, link it, and run it. You can use either DEBUG or DOS to run a program. Use DEBUG if the program does not produce visible results.

If your assembler disk is in drive A and your data disk is in drive B, and B is active, the general command forms are:

a:edlin *progname*.**asm**	(edit)
a:masm *progname*;	(assemble)
a:link *progname*;	(link)
a:debug *progname*.**exe**	(run under DEBUG)
progname.**exe**	(run under DOS)

To link several modules, list their names, separated with + signs (e.g., **link mod1+mod2**).

Study Exercises (answers on page 362)

1. How many *bytes* does the following sequence allocate in memory?

```
VAR1   DB   ?
VAR2   DW   4 DUP(?),20
VAR3   DB   10 DUP(?)
```

2. What value does the assembler put into VAR1 in Exercise 1?
3. How do the following statements differ?

```
K   EQU   1024
K   =     1024
```

4. What is wrong with this sequence:

```
CONST   DB      ?
        MOV     CONST,256
```

5. Which pseudo-ops mark the beginning and end of every procedure?
6. What is the difference between a NEAR procedure and a FAR procedure?
7. The first procedure the 8088 executes in every program must be a FAR one. Why?
8. What does the following ASSUME statement do?

```
ASSUME CS:CSEG
```

9. What does the Linker program do?

3

The 8088 Instruction Set

3.1 About This Chapter

In Chapter 2 you began working with the "foundation" of assembly language—the instructions that control the microprocessor. This chapter provides a detailed description of the 8088's (and 8086's) instruction set and operand addressing modes.

Many books cover the instructions individually, discussing them one by one, in alphabetical order. Although that approach has definite merit in a technical reference manual, it tends to leave you bored and bewildered after the fifth or sixth instruction.

In this book we group instructions by function, so similar instructions are described together. That is, we group add instructions with subtract instructions, shifts with rotates, and so on. This should help you *understand* the instruction set, and how individual instructions "fit together," so you don't learn them as just a lot of disjointed entities.

Later, after running a few programs, you will need to refer to this chapter only occasionally, to look up details of specific instructions. Once you feel comfortable with the instruction set, you can answer most questions by referring to Appendix D, where the instructions are summarized alphabetically. Appendix C is also useful; it tells you how long each instruction takes to execute.

3.2 Addressing Modes

The 8088 provides a variety of ways to access the operands your programs are to operate on. Operands can be contained in registers, within the instruction itself, or in memory or an I/O port. Some of the manufacturers' marketing literature claims that the 8088 has 24 operand addressing modes—perhaps justifiably, if you consider all the possible operand

combinations. In this book, however, we divide the addressing modes into seven groups:

1. Register addressing
2. Immediate addressing
3. Direct addressing
4. Register indirect addressing
5. Base relative addressing
6. Direct indexed addressing
7. Base indexed addressing

The microprocessor determines *which* of the seven addressing modes to use by examining the contents of a "mode field" in the instruction. The assembler sets up the mode field based on how the operand(s) appear in your source program. For instance, if you write

```
MOV  AX,BX
```

the assembler encodes both operands (*AX* and *BX*) for the register addressing mode. However, if you put brackets around the source operand, and write

```
MOV AX,[BX]
```

the assembler encodes the source operand (BX) for the register *indirect* addressing mode.

Table 3-1 shows the assembler format, and which segment register is used to calculate the physical address, for the 8088's seven operand addressing modes. Note that all modes assume you are accessing the data segment (DS is the segment register) except those that involve BP, in which case the stack segment is assumed (SS is the segment register).

Important: The 8088's string instructions (described in Section 3.8) assume that DI points to a location in the extra segment, rather than the data segment. Hence, they use ES as the segment register. All other instructions follow the assignments in Table 3-1.

Of the seven addressing modes, the 8088 processes register and immediate operands fastest, because its Execution Unit (EU) can fetch them from registers (for register addressing) or from the instruction pipeline (for immediate addressing). The other modes take longer because the Bus Interface Unit (BIU) must compute a memory address, then fetch the operand and pass it to the EU.

Table 3-1. 8088 addressing modes.

Addressing Mode	*Operand Format*	*Segment Register*
Register	reg	None
Immediate	data	None
Direct	disp	DS
	label	DS
Register indirect	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Base relative	[BX] + disp	DS
	[BP] + disp	SS
Direct indexed	[DI] + disp	DS
	[SI] + disp	DS
Base indexed	[BX][SI] + disp	DS
	[BX][DI] + disp	DS
	[BP][SI] + disp	SS
	[BP][DI] + disp	SS

Notes: 1. *disp* is optional for base indexed addressing.
2. *reg* can be any 8- or 16-bit register, except IP.
3. *data* can be an 8- or 16-bit constant value.
4. *disp* can be an 8- or 16-bit signed displacement value.

Each addressing mode description in this section includes an example of the mode's usage. In most cases, we use the 8088's *MOV* (Move) instruction to demonstrate the mode.

Register and Immediate Addressing

In *register addressing*, the 8088 fetches an operand from (or loads it into) a register. For example, the instruction

```
MOV  AX,CX
```

copies the 16-bit contents of the count register (CX) into the accumulator register (AX). The contents of CX are unaffected. In this example, the 8088 uses register addressing to fetch the source operand from CX and to load it into the destination register, AX.

Immediate addressing lets you specify an 8- or 16-bit constant value as a source operand. This constant is contained in the instruction (where it was put by the assembler), rather than in a register or memory location. For example,

```
MOV   CX,500
```

loads the value 500 into the CX register and

```
MOV   CL,-30
```

loads -30 into the CL register.

The immediate operand may also be a symbol that was defined by an EQU directive, so this kind of form is valid:

```
K   EQU   1024
    ..
    ..
    MOV   CX,K
```

To avoid problems, remember that 8-bit signed numbers can range from 127 (7FH) to -128 (80H), and 16-bit signed numbers can range from 32767 (7FFFH) to -32768 (8000H). The maximum 8- and 16-bit unsigned values are 255 (0FFH) and 65535 (0FFFFH), respectively.

Immediate Values Are Sign-Extended

The assembler always sign-extends immediate values in the destination. This means it repeats the most-significant bit of the source value to fill the 8 or 16 bits of the destination operand.

For instance, the source operand for our first example, decimal 500, can be represented by the 10-bit binary pattern 0111110100. When the assembler realizes that you are loading this value into a 16-bit register (CX), it extends the pattern to 16 bits by preceding it with six copies of the "sign" bit value (0). Therefore, CX will end up with the binary pattern *000000*0111110100. In the second example, the 8088 loads the 8-bit binary pattern for -30 (11100010) into CL.

Memory Addressing Modes

As we mentioned in Chapter 1, accessing memory involves a joint effort by the 8088's Execution Unit (EU) and Bus Interface Unit (BIU). When the EU needs to read or write a memory operand, it passes an offset value to the BIU. The BIU adds this offset to the contents of a segment register (with four zeros(0) appended) to produce a 20-bit physical address, then uses that address to access the operand.

The Effective Address

The offset that the Execution Unit calculates for a memory operand is called the operand's *effective address* (EA). The EA is the operand's distance in bytes from the beginning of the segment it is in. Being a 16-bit unsigned value, the EA can reference operands that lie up to 65,535 (or 64K) bytes beyond the beginning of the segment.

The amount of time the Execution Unit takes to calculate the EA is one of the prime factors in determining how long an instruction takes to execute. Depending on which addressing mode is used, deriving the EA may involve nothing more than fetching a displacement from within the instruction. Then again, it may require some lengthy calculation, such as adding a displacement, a base register, and an index register. Even if execution time is not critical in your programs, you should appreciate these time factors as you read the addressing mode descriptions that follow.

Direct Addressing

With direct addressing, the EA is contained in the instruction (just as immediate data values are contained in instructions). The 8088 adds the EA to the (shifted) contents of the Data Segment (DS) register to produce the operand's 20-bit physical address.

The direct addressing operand is generally a label. For example, the instruction

```
MOV   AX,TABLE
```

loads the contents of data memory location TABLE into the AX register. Figure 3-1 shows how this instruction works.

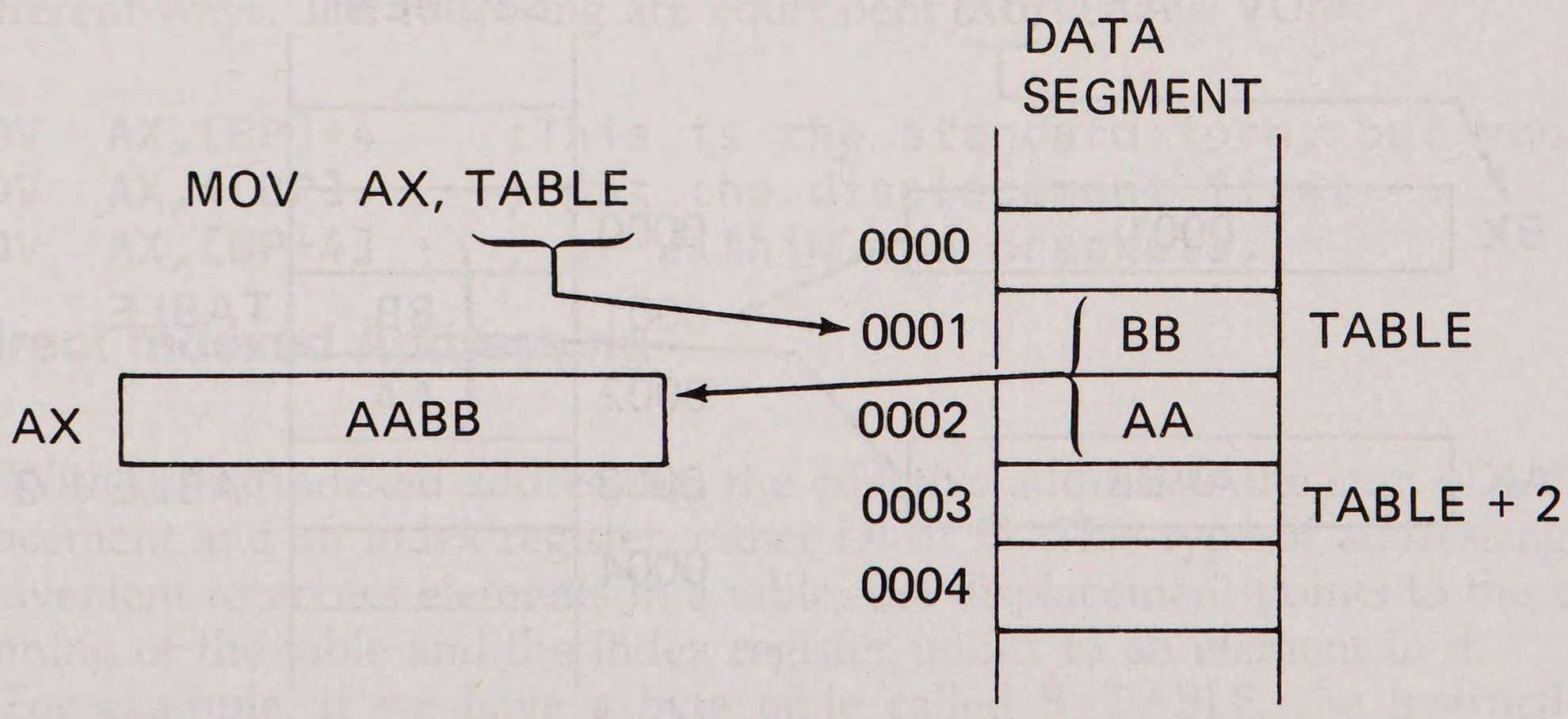


Figure 3-1. Direct addressing.

Note that the 8088 stores data in memory in the reverse order you would expect: with the high-order byte *following* (rather than preceding) the low-order byte. To keep this straight, just remember that *the high (most-significant) part of the data is in the highest memory address.*

Register Indirect Addressing

With register indirect addressing, the effective address of the operand is contained in base register BX, base pointer BP, or an index register (SI or DI). You must enclose register indirect operands in square brackets to distinguish them from register operands. For example, the instruction

```
MOV   AX,[BX]
```

loads the contents of the memory location addressed by BX into the AX register. Figure 3-2 illustrates this example.

How do you put an offset address into BX? One way is by applying the OFFSET operator to the memory address. For example, to load the word at location *TABLE* into AX, use the sequence

```
MOV   BX,OFFSET TABLE
MOV   AX,[BX]
```

These two instructions do the same job as the single instruction

```
MOV   AX,TABLE
```

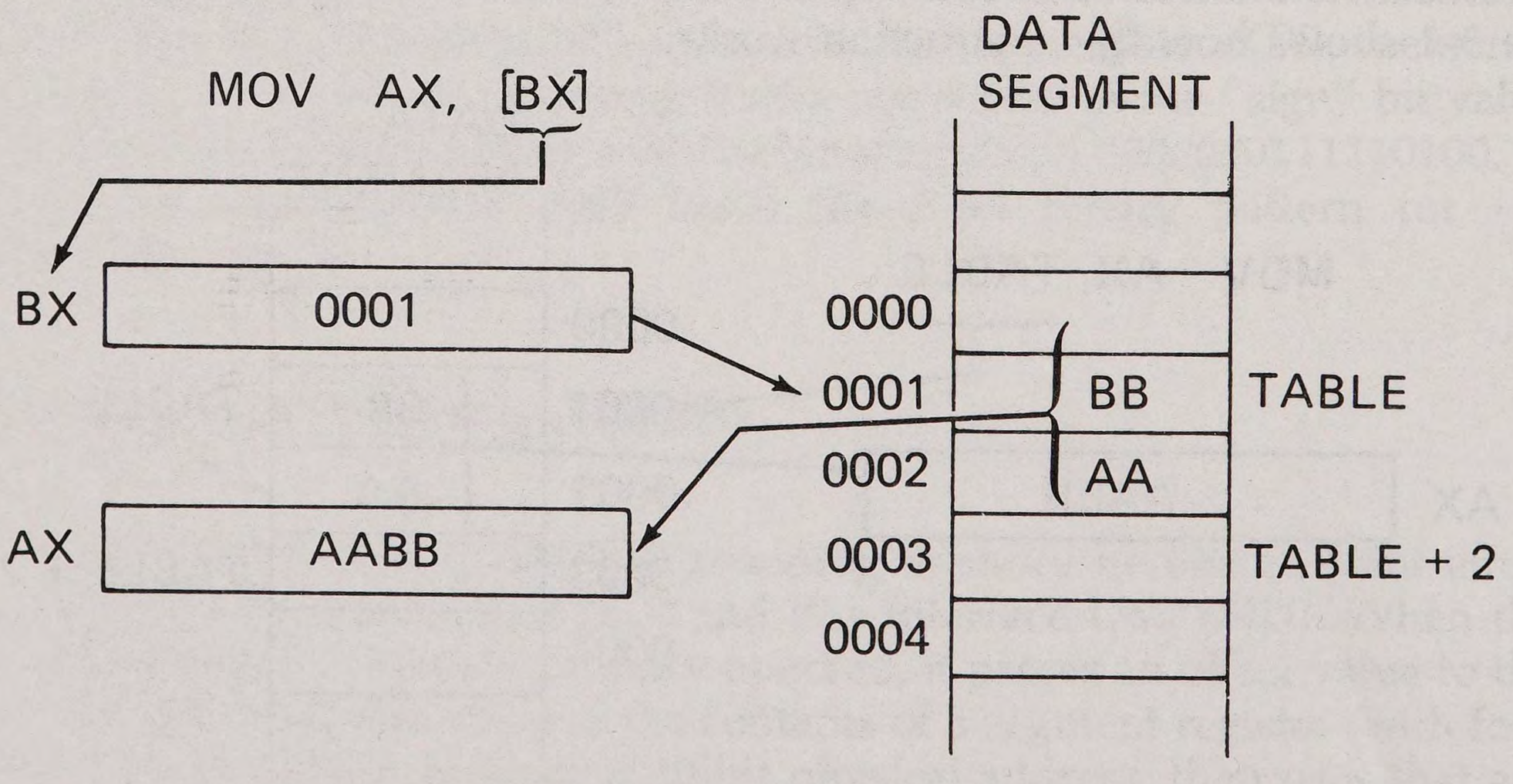


Figure 3-2. Register indirect addressing.

except that they destroy the previous contents of BX. If you plan to access just one memory location (the contents of TABLE here), this single-instruction approach makes more sense. However, to access *several* locations, starting at that base address, having the effective address in a register is better. Why? Because you can manipulate the contents of the register without fetching a new address each time.

Base Relative Addressing

With base relative addressing, the assembler calculates the effective address by adding a displacement value to the contents of the BX or BP register.

The BX form gives you a convenient way to access data structures that are located in different parts of memory. To do this, you put the base address of the structure into the base register and reference elements of the structure by their displacement from the base. After that, you can access different records in the structure by simply changing the base register.

For example, suppose you have read some personnel records from disk. Each record contains an employee's identification number, department number, division number, age, pay rate, etc. If the division number is stored in the fifth and sixth bytes of the record, and the starting address of the record is in BX, the instruction

```
MOV AX,[BX]+4
```

loads the employee's division number into AX. (The displacement is 4, rather than 5, because the first byte is Byte 0.) Figure 3-3 illustrates this example.

The IBM assembler lets you specify the base relative operand in three different ways. The following are equivalent instructions:

```
MOV  AX,[BP]+4     ;This is the standard form, but you may
MOV  AX,4[BP]      ; put the displacement first
MOV  AX,[BP+4]     ; or within the brackets.
```

Direct Indexed Addressing

With direct indexed addressing, the effective address is the sum of a displacement and an index register, either DI or SI. This type of addressing is convenient to access elements in a table; the displacement points to the beginning of the table and the index register points to an element in it.

For example, if we have a byte table called B_TABLE, the instruction sequence

```
MOV   DI,2
MOV   AL,B_TABLE[DI]
```

loads the third element into the AL register.

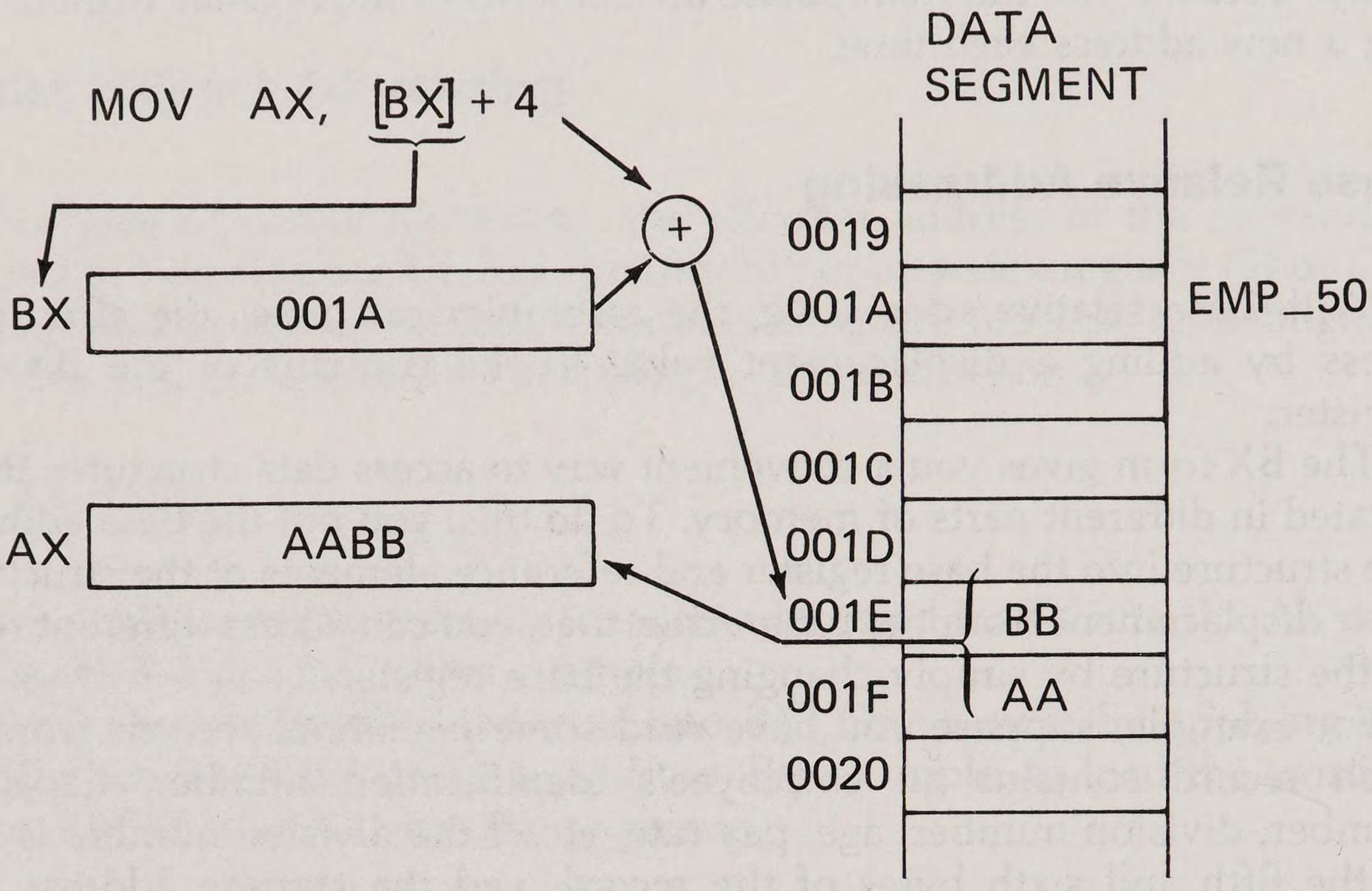


Figure 3-3. Base relative addressing.

In a word table, the elements are two bytes apart, so you must *double the element number* to get its index value. With a word table called TABLE, the instruction sequence

```
MOV   DI,4
MOV   AX,TABLE[DI]
```

loads the third element into the AX register. Figure 3-4 illustrates this example.

Base Indexed Addressing

With base indexed addressing, the EA is the sum of a base register, an index register, and (optionally) a displacement.

Because it accepts two separate offsets, this mode is useful for accessing two-dimensional arrays. Here, the base register holds the starting address of the array, and the displacement and index register provide row and column offsets.

For example, suppose your computer monitors six pressure valves in a chemical processing plant. This system reads the valve settings every half-

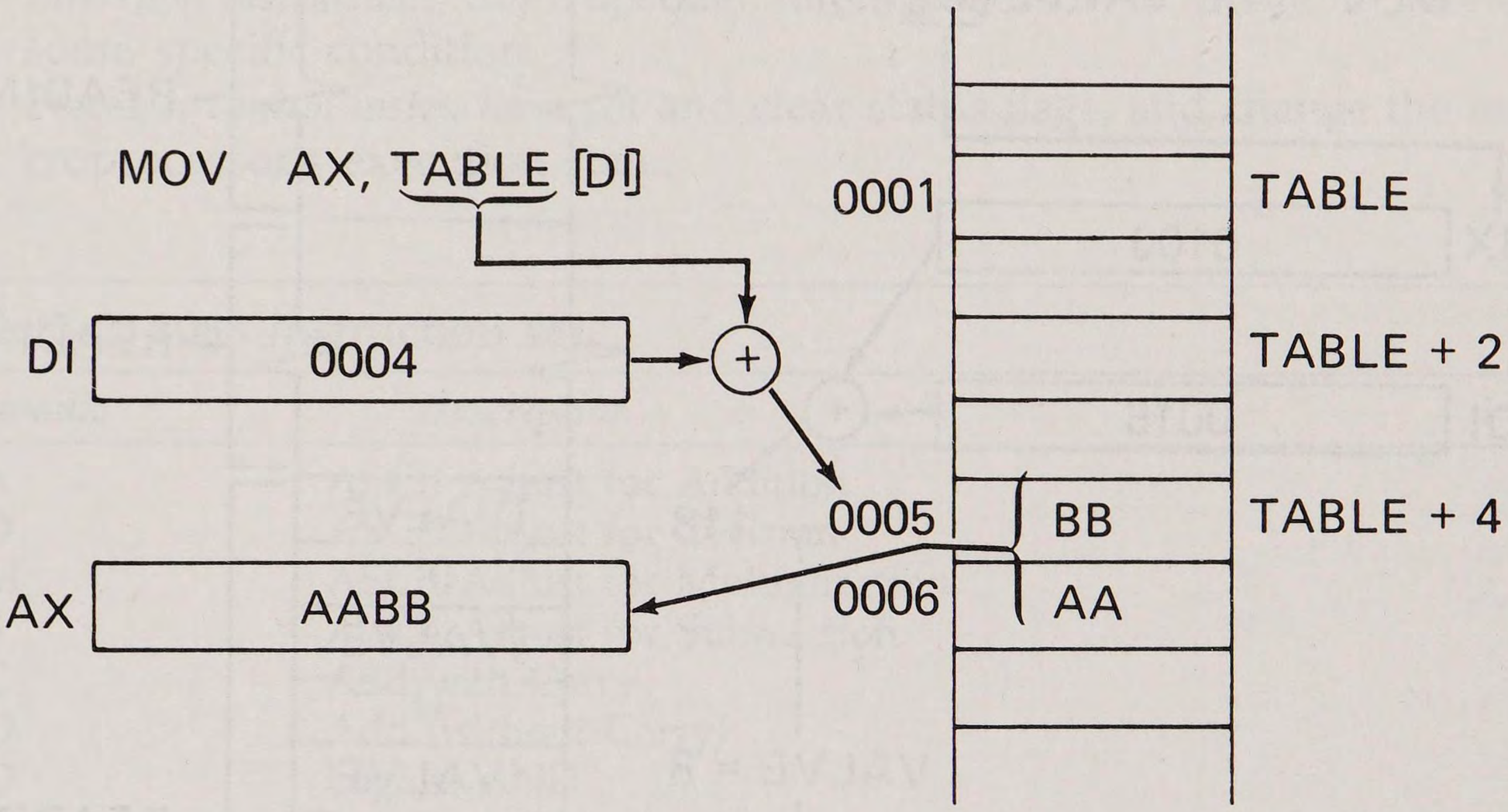


Figure 3-4. Direct indexed addressing.

hour and records them in memory. In one week, these readings form an array that has 336 blocks (48 readings per day for seven days) of six elements each, a total of 2,016 data values.

If the starting address of the array is in BX, the block displacement (reading number times 12) is in DI, and the valve number displacement is defined by the variable VALVE, you can use the instruction

```
MOV  AX,VALVE[BX][DI]
```

to load any selected pressure valve reading into AX. In Figure 3-5, this instruction extracts the third reading (reading 2) of Valve 4 from an array that has a data segment offset of 100H.

Here are some other legal formats for base indexed addressing operands:

```
MOV  AX,[BX+2+DI]   ;The operands can be put in brackets
MOV  AX,[DI+BX+2]   ; in any order
MOV  AX,[BX+2][DI]  ; and the displacement can be paired
MOV  AX,[BX][DI+2]  ; with either register.
```

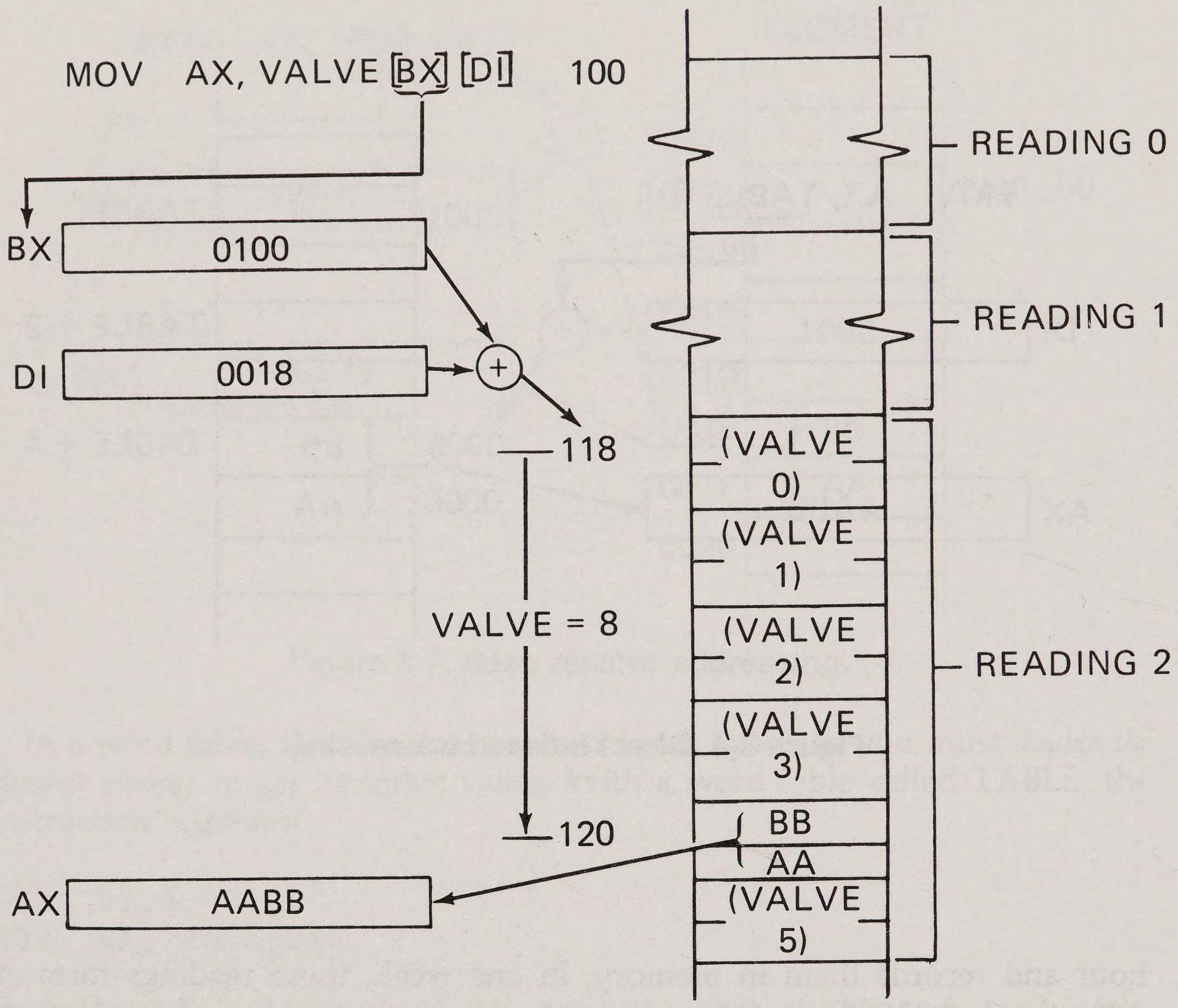


Figure 3-5. Extracting a data value from a two-dimensional array.

3.3 Instruction Types

As we mentioned earlier, the 8088 has 92 instruction types. Table 3-2 shows their assembler mnemonics and tells you what each mnemonic stands for. Note that some instructions have several valid mnemonics.

We can divide the instruction set into seven functional groups:

1. *Data transfer instructions* move information between registers and memory locations or I/O ports.

2. *Arithmetic instructions* perform arithmetic operations on binary or binary-coded decimal (BCD) numbers.
3. *Bit manipulation instructions* perform shift, rotate, and logical operations on memory locations and registers.
4. *Control transfer instructions* control the sequence in which a program executes; they include jumps and transfers to and from procedures.
5. *String instructions* move, compare, and scan strings of information.
6. *Interrupt instructions* interrupt the microprocessor to make it service some specific condition.
7. *Processor control instructions* set and clear status flags, and change the microprocessor's execution state.

Table 3-2. 8088 instruction set.

Mnemonic	*Description*
AAA	ASCII Adjust for Addition
AAD	ASCII Adjust for Division
AAM	ASCII Adjust for Multiplication
AAS	ASCII Adjust for Subtraction
ADC	Add with Carry
ADD	Add (without Carry)
AND	Logical AND
CALL	Call a Procedure
CBW	Convert Byte to Word
CLC	Clear Carry Flag
CLD	Clear Direction Flag
CLI	Clear Interrupt Flag
CMC	Complement Carry Flag
CMP	Compare Destination to Source
CMPS, CMPSB, or CMPSW	Compare Byte or Word Strings
CWD	Convert Word to Doubleword
DAA	Decimal Adjust for Addition
DAS	Decimal Adjust for Subtraction
DEC	Decrement Destination by One
DIV	Divide, Unsigned
ESC	Escape
HLT	Halt the Processor
IDIV	Integer Divide, Signed
IMUL	Integer Multiply, Signed
IN	Input Byte or Word
INC	Increment Destination by One
INT	Interrupt
INTO	Interrupt If Overflow

Table 3-2. 8088 instruction set (continued).

Mnemonic	*Description*
IRET	Interrupt Return
JA or JNBE	Jump If Above/If Not Below nor Equal
JAE, JNB, or JNC	Jump If Above or Equal/If Not Below/If No Carry
JB, JNAE, or JC	Jump If Below/If Not Above nor Equal/If Carry
JBE or JNA	Jump If Below or Equal/If Not Above
JCXZ	Jump If CX Is Zero
JE or JZ	Jump If Equal/If Zero
JG or JNLE	Jump If Greater/If Not Less nor Equal
JGE or JNL	Jump If Greater or Equal/If Not Less
JL or JNGE	Jump If Less/If Not Greater Nor Equal
JLE or JNG	Jump If Less or Equal/If Not Greater
JMP	Jump Unconditionally
JNE or JNZ	Jump If Not Equal/If Not Zero
JNO	Jump If No Overflow
JNP or JPO	Jump If No Parity/If Parity Odd
JNS	Jump If No Sign (If Positive)
JO	Jump On Overflow
JP or JPE	Jump On Parity/If Parity Even
JS	Jump On Sign
LAHF	Load AH from Flags
LDS	Load Pointer Using DS
LEA	Load Effective Address
LES	Load Pointer Using ES
LOCK	Lock the Bus
LODS, LODSB, or LODSW	Load Byte or Word String
LOOP	Loop Until Count Complete
LOOPE or LOOPZ	Loop While Equal/While Zero
LOOPNE or LOOPNZ	Loop While Not Equal/While Not Zero
MOV	Move
MOVS, MOVSB, or MOVSW	Move Byte or Word String
MUL	Multiply, Unsigned
NEG	Negate (Form Two's-Complement)
NOP	No Operation
NOT	Logical NOT
OR	Logical Inclusive-OR
OUT	Output Byte or Word

Table 3-2. 8088 instruction set (continued).

Mnemonic	*Description*
POP	Pop Word Off Stack to Destination
POPF	Pop Flags Off Stack
PUSH	Push Word onto Stack
PUSHF	Push Flags onto Stack
RCL	Rotate Left through Carry
RCR	Rotate Right through Carry
REP, REPE, or **REPZ**	Repeat String Operation/While Equal/While Zero
REPNE or **REPNZ**	Repeat String Operation While Not Equal/While Not Zero
RET	Return from Procedure
ROL	Rotate Left
ROR	Rotate Right
SAHF	Store AH into Flags
SAL or **SHL**	Shift Arithmetic Left/Logical Left
SAR	Shift Arithmetic Right
SBB	Subtract with Borrow
SCAS, SCASB, or **SCASW**	Scan Byte or Word String
SHR	Shift Logical Right
STC	Set Carry Flag
STD	Set Direction Flag
STI	Set Interrupt Enable Flag
STOS, STOSB, or **STOSW**	Store Byte or Word String
SUB	Subtract (Without Borrow)
TEST	Test (Logically Compare Two Operands)
WAIT	Wait
XCHG	Exchange Two Operands
XLAT	Translate
XOR	Logical Exclusive-OR

In the following sections we will describe the 8088 instruction set by groups, in the order we just presented. Let us begin with the data transfer group, which includes the ubiquitous MOV instruction.

3.4 Data Transfer Instructions

Data transfer instructions move data and addresses between registers and memory locations or I/O ports. Table 3-3 summarizes these instructions in

four groups: general-purpose, input/output, address transfer, and flag transfer.

Table 3-3. Data transfer instructions.

			Flags								
Mnemonic	***Assembler Format***		***OF***	***DF***	***IF***	***TF***	***SF***	***ZF***	***AF***	***PF***	***CF***
General-Purpose											
MOV	MOV	destination,source	-	-	-	-	-	-	-	-	-
PUSH	PUSH	source	-	-	-	-	-	-	-	-	-
POP	POP	destination	-	-	-	-	-	-	-	-	-
XCHG	XCHG	destination,source	-	-	-	-	-	-	-	-	-
XLAT	XLAT	source-table	-	-	-	-	-	-	-	-	-
Input/Output											
IN	IN	accumulator,port	-	-	-	-	-	-	-	-	-
OUT	OUT	port,accumulator	-	-	-	-	-	-	-	-	-
Address Transfer											
LEA	LEA	reg16,mem16	-	-	-	-	-	-	-	-	-
LDS	LDS	reg16,mem32	-	-	-	-	-	-	-	-	-
LES	LES	reg16,mem32	-	-	-	-	-	-	-	-	-
Flag Transfer											
LAHF	LAHF		-	-	-	-	-	-	-	-	-
SAHF	SAHF		-	-	-	-	*	*	*	*	*
PUSHF	PUSHF		-	-	-	-	-	-	-	-	-
POPF	POPF		*	*	*	*	*	*	*	*	*

Note: * means changed and - means unchanged.

General-Purpose Instructions

Move (MOV)

The fundamental general-purpose instruction, *move (MOV)*, can transfer byte or word data between a register and a memory location, or between two registers. It can also transfer an immediate data value into a register or memory location.

The Move instruction has the general form

```
MOV  destination,source
```

Most operand combinations are legal. Here are some examples:

```
MOV   AX,TABLE      ;Move from memory into a register
MOV   TABLE,AX      ; or vice versa.
MOV   ES:[BX],AX    ;A segment override may be included.
MOV   DS,AX         ;Move between 16-bit registers
MOV   BL,AL         ; or 8-bit registers.
MOV   CL,-30        ;Move a constant into a register
MOV   DEST,25H      ; or into memory.
```

The Move instruction *excludes* these combinations:

1. You cannot move data between two memory locations directly. To do this, move the source data into a general-purpose register, then move *that* register into the destination. For example, if POUNDS and WEIGHT are variables in memory, give them the same value with:

   ```
   MOV   AX,POUNDS
   MOV   WEIGHT,AX
   ```

2. You cannot load an immediate value into a segment register. As with rule 1, load it via a general-purpose register. For example, the following loads the segment number of a data segment (DATA_SEG here) into DS:

   ```
   MOV   AX,DATA_SEG
   MOV   DS,AX
   ```

   These kinds of instructions follow the ASSUME statement in a code segment. They tell the assembler where the data segment is located.
3. You cannot move the contents of one segment register into another directly. Make the move via a general-purpose register. For example, to make DS point to the same segment as ES, use

   ```
   MOV   AX,ES
   MOV   DS,AX
   ```

   (You can also use PUSH and POP instructions to perform this operation, as you will see in the next section.)
4. You cannot use the CS register as the destination of a move instruction.

Push (PUSH) and Pop (POP)

We mentioned earlier that the stack holds return addresses while a procedure is being executed. The call (CALL) instruction puts an address onto the

stack and a return (RET) instruction retrieves it at the conclusion of the procedure. This is one of the ways the 8088 uses the stack *automatically,* without you telling it to do so.

The stack is also a convenient place to temporarily deposit data—register and memory operands—from your program. For example, you might want to save the contents of the AX register while you put AX to some other use. Two instructions that let you use the stack are *Push Word onto Stack (PUSH)* and *Pop Word Off Stack (POP).*

PUSH puts the contents of a word-sized (16-bit) register or memory operand on the top of the stack. Conversely, POP retrieves a word from the top of the stack and puts it into a memory location or register.

The PUSH and POP instructions have these general formats:

```
PUSH  source
POP   destination
```

Here are some examples:

```
PUSH  SI                ;You can save a general-purpose register
PUSH  DS                ; or a segment register,
PUSH  CS                ; including CS.
PUSH  COUNTER           ;You can also save the contents of
PUSH  TABLE[BX][DI]     ; a memory location.
```

Being complementary instructions, PUSH and POP are generally used in pairs. That is, for each PUSH in a program there must be a POP. For example, to save the contents of AX on the stack and later restore them, your program has this form:

```
PUSH  AX  ;Save AX on top of the stack.
..              (Other operations are being performed
..               with AX here.)
POP   AX  ;Retrieve AX from top of the stack.
```

By "top" of the stack, we mean the location in the stack segment the stack pointer (SP) is pointing to. Since the stack "builds" downward in memory (toward location 0), the first word pushed onto the stack is stored at the highest stack address, the next is stored two bytes lower, and so on.

The SP always points to the word that was last pushed onto the stack. Therefore, PUSH subtracts 2 from the stack pointer, then transfers the source operand (a word) onto the stack. Conversely, POP transfers the word addressed by SP to the destination operand, then adds 2 to the SP.

Figure 3-6 shows the stack and the stack pointer before and after a PUSH, and after a POP. Following the PUSH (Figure 3-6(B)), the stack pointer points two bytes lower in memory and those previously-unused bytes now

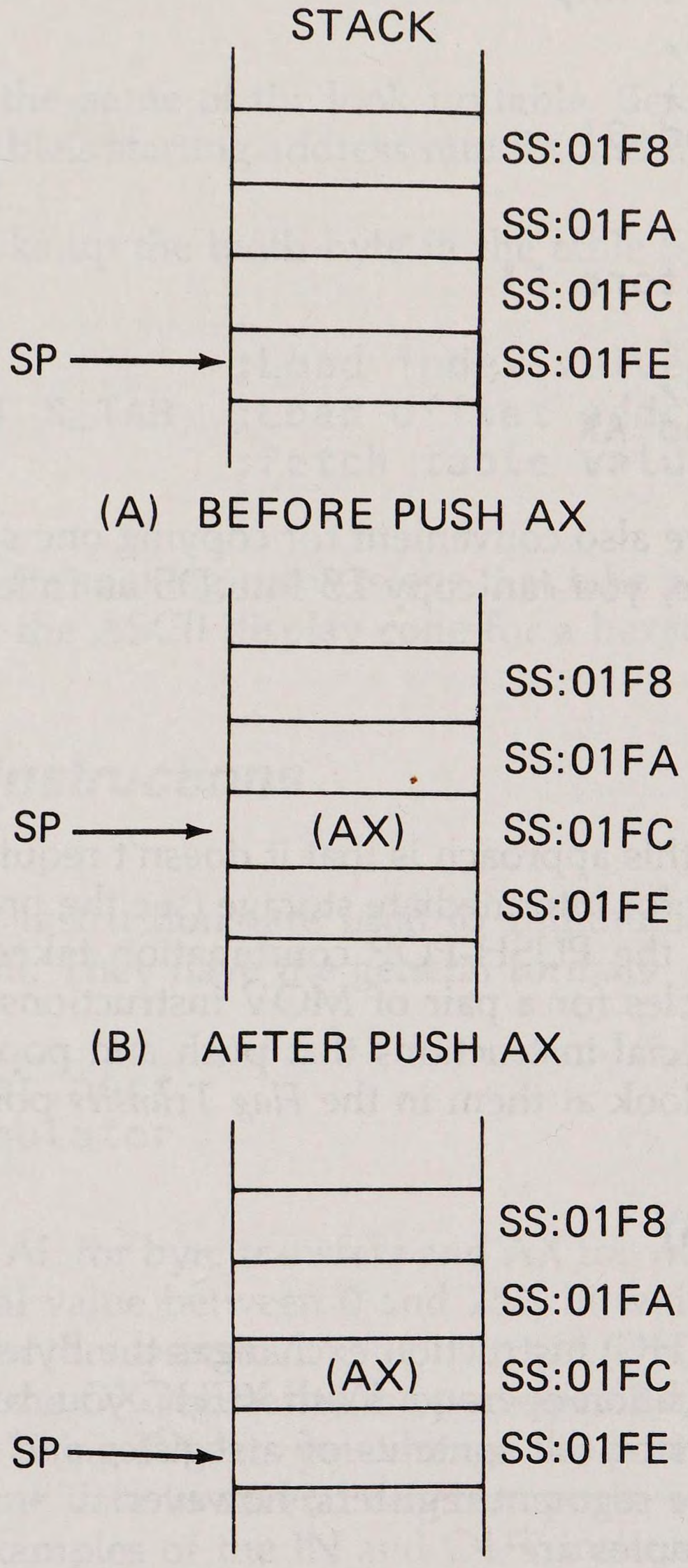


Figure 3-6. How a PUSH and a POP affect the stack.

hold the contents of AX. Following the POP (Figure 3-6(C)), the SP has its original value. (Note that although AX is still in memory, it is not "on the stack.")

You can save more than one word value on a stack by doing a series of pushes. However, remember that because each PUSH puts its data on top of the stack, *you must POP words off the stack in the reverse order you PUSHed them onto it*. The following sequence pushes the contents of four registers onto the stack, and later restores them:

```
PUSH   AX     ;Save AX,
PUSH   ES     ; ES,
PUSH   DI     ; DI,
PUSH   SI     ; and SI
..
..
POP    SI     ;Restore SI
POP    DI     ; DI,
POP    ES     ; ES,
POP    AX     ; and AX
```

PUSH and POP are also convenient for copying one segment register into another. For example, you can copy ES into DS as follows:

```
PUSH   ES
POP    DS
```

The advantage of this approach is that it doesn't require you to use a general-purpose register for intermediate storage (see the preceding section); the disadvantage is that the PUSH-POP combination takes 26 clock cycles to execute, versus 4 cycles for a pair of MOV instructions.

There are also special instructions that push and pop the contents of the flags register. We'll look at them in the *Flag Transfer* portion of this section.

Exchange (XCHG)

The exchange (XCHG) instruction exchanges the byte or word source operand with the destination operand. With XCHG you can exchange the contents of two registers or the contents of a register and a memory location. You cannot exchange segment registers, however.

Some XCHG examples are:

```
XCHG   AX,BX          ;Exchange two word registers
XCHG   AL,BH          ; or two byte registers.
XCHG   WORD_LOC,DX    ;Exchange a memory location
XCHG   DL,BYTE_LOC    ; and a register.
```

Translate (XLAT)

The translate (XLAT) instruction looks up a value in a byte table and loads it into AL. The table may be up to 256 bytes long.

XLAT's general format is

```
XLAT  source-table
```

where *source-table* is the name of the look-up table. Before executing XLAT, you must load the table's starting address into BX and the index value of the desired byte into AL.

This sequence looks up the tenth byte in the table S_TAB:

```
MOV   AL,10                ;Load index value into AL.
MOV   BX,OFFSET S_TAB      ;Load offset address into BX.
XLAT  S_TAB                ;Fetch table value into AL.
```

XLAT is convenient for making conversions that take a lot of time to calculate, such as finding the ASCII display code for a hexadecimal digit.

Input/Output Instructions

The input/output instructions are used to communicate with peripheral devices in the system. They have the general formats

```
IN    accumulator,port
OUT   port,accumulator
```

where *accumulator* is AL for byte transfers and AX for word transfers. Generally, *port* is a decimal value between 0 and 255, in which case it references one of 256 devices in the system.

You may also use the DX register as the port operand, giving you access to 64K different ports. Using DX lets you change the port number easily, perhaps to send the same data to several different ports.

Here are a few examples of the IN and OUT instructions:

```
IN    AL,200           ;Input a byte from Port 200
IN    AL,PORT_VAL      ; or from a port named by a constant.
OUT   30H,AX           ;Output a word to Port 30H
OUT   DX,AX            ; or to the port specified by DX.
```

Address Transfer Instructions

Address transfer instructions transfer the *addresses* of variables rather than their *contents*.

Load Effective Address (LEA)

LEA transfers the offset address of a memory operand into any 16-bit general, pointer, or index register. It has the general format

```
LEA   reg16,mem16
```

where *mem16* must have a type attribute of WORD.

Unlike MOV with an OFFSET operator, the memory operand for LEA can be subscripted, which gives you a lot more addressing flexibility. For example, if the DI register contains 5, the instruction

```
LEA   BX,TABLE[DI]
```

loads the offset address of TABLE+5 into BX.

We will discuss LEA instructions extensively in Section 3.8, where we perform operations on strings.

Load Pointer Using DS (LDS)

The LDS instruction reads a 32-bit doubleword from memory, and loads the first 16 bits into a specified register and the second 16 bits into DS. The general format is

```
LDS   reg16,mem32
```

where *reg16* is any 16-bit general register and *mem32* is a memory location with a DOUBLEWORD type attribute.

Generally, the *mem32* operand is defined with the Define Doubleword (DD) data definition directive we discussed in Section 2.5. Using the example in that section,

```
HERE_FAR   DD   HERE
```

we can get the offset address and segment number of HERE into BX and DS, respectively, with the instruction

```
LDS   BX,HERE_FAR
```

Note that LDS is a one-instruction replacement for a sequence of this kind:

```
MOV   BX,OFFSET HERE
MOV   AX,SEG HERE
MOV   DS,AX
```

Note also that LDS eliminates the need for a third 16-bit register (AX, here).

Load Pointer Using ES (LES)

The LES instruction is identical to LDS, except LES transfers the segment part of the address into ES instead of DS.

Flag Transfer Instructions

Load AH from Flags (LAHF) and Store AH into Flags (SAHF)

The LAHF instruction copies the flags register's 8080/8085 flags into AH. Specifically, it copies CF, PF, AF, ZF, and SF into the corresponding bits of AH (0, 2, 4, 6, and 7, respectively). SAHF performs the reverse operation; it loads the five bits of AH we just mentioned into the flags register.

LAHF does not affect any flags but, of course, SAHF affects the 8080/8085 flags. Both instructions are provided for 8080/8085 compatibility.

Push Flags onto Stack (PUSHF) and Pop Flags Off Stack (POPF)

These instructions transfer the contents of the flags register to and from the stack. They are essentially the same as the PUSH and POP instructions, except PUSHF and POPF require no operands; they reference the flags register implicitly.

As with PUSH and POP, you must always use PUSHF and POPF in pairs. That is, every PUSHF must have a corresponding POPF later in the program, like this:

```
PUSHF           ;Save flags on stack.
..              (Other, flag-altering instructions are executed here)
..
POPF            ;Restore flags from stack.
```

Note that with PUSH, PUSHF, POP, and POPF, you can save any or all registers while a procedure or interrupt routine is executing. For example, suppose you have valuable data in AX, DI, and SI, and need to call a procedure named SORT that uses these registers. Suppose also that you have just performed an arithmetic operation, and need to have the flags intact after the procedure. This sequence does the job:

```
PUSH    AX      ;Save the three registers
PUSH    DI
PUSH    SI
PUSHF           ; and the flags.
CALL    SORT    ;Call the procedure.
POPF            ;Upon return, restore the flags
POP     SI      ; and the three registers.
POP     DI
POP     AX
```

Better yet, put the required PUSHes and POPs *inside* the procedure, so you don't have to repeat them every time you call it. Hence, for the preceding example, the four PUSHes and four POPs should be the first and last instructions in the SORT procedure. Then you can simply CALL SORT without worrying about which registers are destroyed or preserved.

3.5 Arithmetic Instructions

The 8088 can perform arithmetic operations on signed or unsigned binary numbers and on unsigned decimal numbers (either packed or unpacked). As Table 3-4 shows, there are instructions for the four standard arithmetic functions—addition, subtraction, multiplication, and division—and two additional instructions that "sign-extend" operands, letting you operate on mixed-size data. Before discussing the instructions themselves, let's look at the kinds of data they can operate on.

Arithmetic Data Formats

Binary Numbers

Binary numbers may be 8 or 16 bits long, and may be either unsigned or signed. In an *unsigned* number, all 8 or 16 bits represent data. Therefore, unsigned numbers can range from 0 to 255 (8 bits) or 65535 (16 bits). In a *signed* number, the high-order bit—7 or 15—specifies the sign of the number; all other bits hold data. Therefore, signed numbers can range from 127 to -128 (8 bits) or from 32767 to -32768 (16 bits).

Decimal Numbers

The 8088 stores decimal numbers as a series of unsigned byte-size values in either "packed" or "unpacked" form. In a packed decimal number, each

byte holds two *binary-coded decimal (BCD)* digits, with the most-significant digit in the upper four bits. Therefore, a packed decimal byte can hold values from 00 to 99.

Table 3-4. Arithmetic instructions.

			Flags								
Mnemonic	*Assembler Format*		*OF*	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
Addition											
ADD	ADD	destination,source	*	-	-	-	*	*	*	*	*
ADC	ADC	destination,source	*	-	-	-	*	*	*	*	*
AAA	AAA		?	-	-	-	?	?	*	?	*
DAA	DAA		?	-	-	-	*	*	*	*	*
INC	INC	destination	*	-	-	-	*	*	*	*	-
Subtraction											
SUB	SUB	destination,source	*	-	-	-	*	*	*	*	*
SBB	SBB	destination,source	*	-	-	-	*	*	*	*	*
AAS	AAS		?	-	-	-	?	?	*	?	*
DAS	DAS		?	-	-	-	*	*	*	*	*
DEC	DEC	destination	*	-	-	-	*	*	*	*	-
NEG	NEG	destination	*	-	-	-	*	*	*	*	*
CMP	CMP	destination,source	*	-	-	-	*	*	*	*	*
Multiplication											
MUL	MUL	source	*	-	-	-	?	?	?	?	*
IMUL	IMUL	source	*	-	-	-	?	?	?	?	*
AAM	AAM		?	-	-	-	*	*	?	*	?
Division											
DIV	DIV	source	?	-	-	-	?	?	?	?	?
IDIV	IDIV	source	?	-	-	-	?	?	?	?	?
AAD	AAD		?	-	-	-	*	*	?	*	?
Sign-Extension											
CBW	CBW		-	-	-	-	-	-	-	-	-
CWD	CWD		-	-	-	-	-	-	-	-	-

Note: * means changed, - means unchanged, and ? means undefined.

In an unpacked decimal number, each byte holds just one BCD digit in the lower four bits. Therefore, an unpacked decimal byte can hold values only from 0 to 9. The upper four bits must be zero for multiplication and division, but may be any value for addition or subtraction.

How does the 8088 know *which* kind of data you are operating on? If you add two bytes, for example, how does it know whether they are signed binary numbers, unsigned binary numbers, packed decimal numbers, or unpacked decimal numbers? The 8088 doesn't know what the data represents, nor does it care; it treats *all* operands as binary numbers.

This is fine if your operands are indeed binary, but if they happen to be decimal the results will be incorrect, obviously. To compensate, the 8088 has a group of "adjust" instructions that make decimal operations give the proper result. We will discuss these instructions at the appropriate places in this section.

How Numbers Are Stored in Memory

As we mentioned earlier, the 8088 stores 16-bit numbers in the opposite order you might expect: with the least-significant byte at the lower address. For example, when storing 1234H at a location called NUM, it puts 34H at NUM and 12H at NUM+1. Keep this convoluted storage scheme in mind when you display (or "dump") the contents of memory. Just remember "low data, low address; high data, high address."

Addition Instructions

Add (ADD) and Add With Carry (ADC)

The ADD and ADC instructions can add either 8- or 16-bit operands. ADD adds a source and destination operand and puts the result in the destination. Symbolically, this can be represented as

```
destination = destination + source
```

ADC does the same thing as ADD, except it includes the Carry Flag (CF) in the addition, like this:

```
destination = destination + source + Carry
```

You may not be familiar with the concept of *carry* as applied to computers, but you have certainly encountered it before in adding decimal numbers with pencil and paper. For instance, the following addition produces two carries:

```
  98
  13
+ 79
-----
 190
```

Adding the "ones" column produces an excess of 2, which gets carried into the "tens" position. Then, adding the tens column along with the carry produces another carry (1), which goes into the "hundreds" column. A carry is produced whenever a column cannot hold the sum of all the digits in it.

Similarly, when the computer adds binary numbers, it generates a carry whenever the destination operand cannot hold a sum. For instance, we know that an 8-bit register can hold unsigned values between 0 and 255 (decimal). If we perform a binary addition of 250 and 10, we get

```
    1111 1010     (Binary representation of 250)
  + 0000 1010     (Binary representation of 10)
-------------
1   0000 0100     (Answer = 260 decimal)
```

The sum is correct, but it takes nine bit positions to represent it! If we are using 8-bit registers, the lower eight bits are returned in the destination register, and the ninth bit is returned in the *Carry Flag (CF)*.

Now you can understand why the 8088 has two separate add instructions. One instruction, ADD, can add single-byte or single-word numbers and the low-order terms of multi-precision numbers. The other, ADC, is used to add the higher-order terms of two multi-precision numbers.

For example,

```
ADD  AX,CX
```

adds the 16-bit contents of AX and CX and returns the result in AX. If your operands are longer than 16 bits, you can use this kind of sequence:

```
ADD  AX,CX    ;Add low-order 16 bits,
ADC  BX,DX    ; then high-order 16 bits.
```

which adds the 32-bit number in CX and DX to the one in AX and BX. Here, the ADC instruction includes any carry out of (CX)+(AX) into (DX)+(BX).

You may also add a memory operand to a general register (or vice versa) or add an immediate value to a register or to memory. Some examples are:

```
ADD  AX,MEM_WORD    ;Add a memory operand to a register
ADD  MEM_WORD,AX    ; or vice versa.
ADD  AL,10          ;Add a constant to a register
ADD  MEM_BYTE,0FH   ; or to a memory location.
```

Most combinations are legal, but you may not add memory-to-memory, nor may you use an immediate value as a destination.

ADD and ADC can affect six flags:

- The Carry Flag (CF) is 1 if the result cannot be contained in the destination operand; otherwise, CF is 0.
- The Parity Flag (PF) is 1 if the result has an even number of 1 bits; otherwise, PF is 0.
- The Auxiliary Carry Flag (AF) is 1 if the result of a decimal addition needs to be adjusted; otherwise, AF is 0.
- The Zero Flag (ZF) is 1 if the result is zero; otherwise, ZF is 0.
- The Sign Flag (SF) is 1 if the result is negative (the high-order bit is a 1); otherwise, SF is 0.
- The Overflow Flag (OF) is 1 if adding two like-signed numbers (both positive or both negative) gives a result that exceeds the two's-complement range of the destination, which changes the sign; otherwise, OF is 0.

SF and OF are pertinent only when you add signed numbers. AF is pertinent only when you add decimal numbers.

The 8088 has instructions that test flags and base an execution decision on the outcome. For instance, a negative result (SF = 1) may cause one set of instructions to execute, while a non-negative result (SF = 0) causes a different set to execute. We discuss these "decision-making" instructions later in the chapter.

ASCII and Decimal Adjust for Addition (AAA and DAA)

As we mentioned earlier, the 8088 performs all additions as if the operands were binary. What happens if they are binary-coded decimal (BCD) numbers? Let's find out by looking at an example. If you add the packed BCD numbers 26 and 55, the 8088 performs this binary addition:

```
  0010 0110     (= BCD 26)
+ 0101 0101     (= BCD 55)
-------------
  0111 1011     (= ??)
```

Instead of the correct answer (BCD 81), the result has a high digit of 7 and a low digit of hexadecimal B. Does this mean you can't add decimal numbers? No, but it means that you must *adjust* the result, to put it into BCD form.

The instructions ASCII Adjust for Addition (AAA) and Decimal Adjust for Addition (DAA) adjust the result of a decimal addition. Neither instruction takes an operand; they both assume that the value that needs adjusting is in the AL register.

AAA converts the contents of AL to a valid *unpacked* decimal digit in the lower four bits of AL (and it puts zeros in the upper four bits). It is used in this context:

```
ADD   AL,BL     ;Add unpacked BCD numbers in AL and BL
AAA             ; and make the result an unpacked number.
```

If the result exceeds 9, AAA adds 1 to AH (to reflect the excess digit) and sets CF to 1; otherwise, it clears CF to 0. AAA also updates AF and leaves PF, ZF, SF, and OF undefined. But since only CF is pertinent, consider these other statuses destroyed.

DAA converts the value in AL to two valid *packed* decimal digits. It is used in this context:

```
ADD   AL,BL     ;Add packed BCD numbers in AL and BL
DAA             ; and make the result a packed number.
```

If the result exceeds the packed BCD limit (99), DAA adds 1 to AH and sets CF to 1. DAA also updates PF, AF, ZF, and SF, and leaves OF undefined. But since only CF is pertinent, consider these other five statuses destroyed.

Increment Destination by One (INC)

The INC instruction adds 1 to a register or memory operand but, unlike ADD, does not affect the Carry Flag (CF). INC is convenient for updating counters in repetitive loop operations. It can also be used to increase an index register or pointer when you are accessing consecutive locations in memory.

Some examples of INC are:

```
INC   CX                ;Increment a word register
INC   AL                ; or a byte register.
INC   MEM_BYTE          ;Increment a byte in memory
INC   MEM_WORD[BX]      ; or a word in memory.
```

Curiously, the 8088 takes longer to increment an 8-bit register than it does a 16-bit register (see the INC entry in Appendix C)! This happens because the Intel designers assumed that programmers would use word-size counters more than byte-size counters, so they provided a special, one-byte version of INC for 16-bit registers.

Subtraction Instructions

How the 8088 Subtracts

Like every other general-purpose microprocessor, the 8088 has no internal subtraction unit. However, it *does* have an *addition* unit—an adder—and can subtract numbers by adding them. Strange as this seems, the concept is "elementary," as Mr. Holmes would say.

To see how to subtract by adding, consider how you subtract 7 from 10. In elementary school you learned to write this as

```
10 − 7
```

However, later (in Algebra 101, perhaps) you learned that another way to write this operation is

```
10 + (−7)
```

The first form—the straight subtraction—could be performed by a processor that has a subtraction unit. Since the 8088 has no such unit, it subtracts in two steps. First, it negates, or *complements*, the subtrahend (the second number). Then it adds the minuend and complemented subtrahend to produce the result. Because the 8088 works with base 2 (binary) numbers, the complement is a *two's-complement*.

To obtain the two's-complement of a binary number, take its positive form and reverse each bit (change each 1 to 0 and each 0 to 1), then add 1 to the result.

Applying this to our "10 − 7" example, the 8-bit binary representations of 10 and 7 are 00001010B and 00000111B, respectively. Take the two's-complement of 7 as follows:

```
  1111 1000      (Reverse all bits)
+          1     (Add 1)
-----------
  1111 1001      (Two's-complement of 7, or -7)
```

Now, the subtraction operation becomes

```
  0000 1010      (=  10)
+ 1111 1001      (=  −7)
-----------
  0000 0011      (Answer = 3)
```

Eureka! We got the right answer!

Since the 8088 does the two's-complementing automatically, there aren't many occasions when you will want to do it yourself. Later in this section we will study an instruction called NEG that takes a two's-complement, in case you ever need it.

Subtract (SUB) and Subtract With Borrow (SBB)

SUB and SBB are similar to their addition counterparts (ADD and ADC), but with subtraction, the Carry Flag (CF) acts as a *borrow* indicator. SUB subtracts a source operand from a destination operand and returns the result in the destination. That is,

```
destination = destination - source
```

SBB does the same thing, except it also subtracts out the Carry Flag (CF), like this:

```
destination = destination - source - Carry
```

As with addition, the subtraction instructions perform two separate functions. One instruction, SUB, subtracts single-byte or single-word numbers, or the low-order terms of multi-precision numbers. The other, SBB, subtracts the higher-order terms of two multi-precision numbers.

For example, the instruction

```
SUB   AX,CX
```

subtracts CX from AX and returns the result in AX.

For operands longer than 16 bits, use this kind of sequence:

```
SUB   AX,CX     ;Subtract low-order 16 bits
SBB   BX,DX     ; then high-order 16 bits.
```

Here we subtract the 32-bit number in CX and DX from the one in AX and BX. SBB includes any borrow out of the first subtraction when it subtracts DX from BX.

You may also subtract a memory operand from a register (or vice versa), or an immediate value from a register or a memory location. The following forms are legal:

```
SUB   AX,MEM_WORD         ;Subtract memory from a register
SUB   MEM_WORD[BX],AX     ; or vice versa.
SUB   AL,10               ;Subtract constant from a register
SUB   MEM_BYTE,0FH        ; or from a memory location.
```

You may not subtract one memory value from another directly, nor may you use an immediate value as a destination.

SUB and SBB can affect six flags:

- The Carry Flag (CF) is 1 if a borrow was needed; otherwise, it is 0.
- The Parity Flag (PF) is 1 if the result has an even number of 1 bits; otherwise, it is 0.
- The Auxiliary Carry Flag (AF) is 1 if the result of a decimal subtraction adjusting; otherwise, it is 0.
- The Zero Flag (ZF) is 1 if the result is zero; otherwise, it is 0.
- The Sign Flag (SF) is 1 if the result is negative (the high-order bit is 1); otherwise, it is 0.
- The Overflow Flag (OF) is 1 if you subtract a positive number from a negative number (or vice versa) and the result exceeds the two's-complement capacity of the destination, which changes the sign; otherwise, OF is 0.

SF or OF are pertinent only when you subtract signed numbers. AF is pertinent only when you subtract decimal numbers.

ASCII and Decimal Adjust for Subtraction (AAS and DAS)

As with addition, the 8088 subtracts as if the operands were binary numbers. This means that your answers may be incorrect if you are subtracting binary-coded decimal (BCD) numbers. For example, suppose you want to subtract BCD 26 from BCD 55. The 8088 performs a binary subtraction by taking the two's-complement of 26, then performs this addition:

```
  0101 0101     (= BCD 55)
+ 1101 1010     (= two's-complement of BCD 26)
-------------
1 0010 1111     (= ??)
```

Instead of the correct answer (BCD 29), the result has a high digit of 2, a low digit of hexadecimal F, and a carry. Clearly, this result sorely needs adjusting.

The instructions ASCII Adjust for Subtraction (AAS) and Decimal Adjust for Subtraction (DAS) adjust the result after you subtract two decimal numbers. Both assume that the number to adjust is in the AL register.

AAS converts the contents of AL to a valid *unpacked* decimal digit in the lower four bits of AL (and it puts zeros in the higher four bits). AAS is used in this context:

```
SUB  AL,BL ;Subtract BCD number in BL from AL
AAS        ; and make the result an unpacked number.
```

If the result exceeds 9, AAS subtracts 1 from AH and sets CF to 1; otherwise, it clears CF to 0. AAS also updates AF and leaves PF, ZF, SF, and OF undefined. But since only CF is pertinent, consider these other statuses destroyed.

DAS converts the contents of AL to two valid *packed* decimal digits. It is used in this context:

```
SUB   AL,BL     ;Subtract packed BCD number in BL from AL
DAS             ; and make the result a packed number.
```

If the result exceeds the packed BCD limit (99), DAS subtracts 1 from AH and sets CF to 1; otherwise, it clears CF to 0. DAS also updates PF, AF, ZF, and SF, and leaves OF undefined. But since only CF is pertinent, consider these other statuses destroyed.

Decrement Destination by One (DEC)

The DEC instruction subtracts 1 from a register or memory operand, but unlike SUB, does not affect the Carry Flag (CF). DEC is often used in loops to decrement a counter until the count becomes zero or negative. It can also be used to decrease an index register or pointer when you access consecutive memory locations.

Some examples are:

```
DEC   CX                ;Decrement a word register
DEC   AL                ; or a byte register.
DEC   MEM_BYTE          ;Decrement a byte in memory
DEC   MEM_WORD[BX]      ; or a word in memory.
```

Negate (NEG)

The NEG instruction subtracts the destination operand from zero, thereby forming the operand's two's-complement. NEG affects the flags in the same way as SUB. However, since one operand is zero, we can be more explicit about the conditions that set individual flags. Therefore, for NEG:

- The Carry Flag (CF) and the Sign Flag (SF) are 1 if the operand is a nonzero positive number; otherwise, they are 0.
- The Parity Flag (PF) is 1 if the result has an even number of 1 bits; otherwise, it is 0.
- The Zero Flag (ZF) is 1 if the operand is zero; otherwise, it is 0.
- The Overflow Flag (OF) is 1 if the operand has the value 80H (byte) or 8000H (word); otherwise, it is 0.

NEG is useful for subtracting a register or memory operand from an immediate value. For instance, you may want to subtract the value in AL from 100. Since an immediate value can't serve as a destination, the form SUB 100,AL is illegal. As an alternative, you can negate AL and *add* 100 to the result, like this:

```
NEG   AL
ADD   AL,100
```

Compare Destination to Source (CMP)

Most programs don't execute instructions in the exact order they are stored in memory. Instead, they usually include jumps, loops, procedure calls, and other factors that make the 8088 transfer from one place in memory to another. We will discuss the instructions that actually produce these transfers later in this chapter, when we discuss the control transfer instructions. At this point we will discuss the CMP instruction, which is commonly used to configure the flags on which the control transfer instructions base their transfer/no-transfer "decisions."

Like the SUB instruction, CMP subtracts a source operand from a destination operand, and sets or clears certain flags based on the result (see Table 3-5). But unlike SUB, *CMP does not save the result of the subtraction.*

That is, CMP doesn't alter the operands. Its sole purpose is to set up the flags for decision-making by conditional jump instructions.

Table 3-5. CMP instruction results.

Condition	*OF*	*SF*	*ZF*	*CF*
Unsigned Operands				
Source < Destination	D	D	0	0
Source = Destination	D	D	1	0
Source > Destination	D	D	0	1
Signed Operands				
Source < Destination	0/1	0	0	D
Source = Destination	0	0	1	D
Source > Destination	0/1	1	0	D

Note: "D" means Don't Care; "0/1" means the flag may be either 0 or 1, depending on the values of the operands.

Multiplication Instructions

Multiply, Unsigned (MUL) and Integer Multiply, Signed (IMUL)

If you have ever endured the agony of writing a multiplication program for the Z80, 6502, or any other conventional 8-bit microprocessor, you will be glad to hear that the 8088 has built-in multiplication instructions. Multiply (MUL) multiplies unsigned numbers and Integer Multiply (IMUL) multiplies signed numbers. Both can multiply bytes or words.

These instructions have the general forms

```
MUL    source
IMUL   source
```

where *source* is a byte- or word-length general register or memory location. For the second operand, MUL and IMUL use the contents of AL (for byte operations) or AX (for word operations). The double-length products are returned as follows:

- A *byte* multiplication returns the 16-bit product in AH (high byte) and AL (low byte).
- A *word* multiplication returns the 32-bit product in DX (high word) and AX (low word).

Upon completion, the Carry Flag (CF) and the Overflow Flag (OF) tell how much of the product is relevant. For MUL, if the high-order half of the product is zero, CF and OF are 0; otherwise, they are 1. For IMUL, if the high-order half of the product is just a sign-extension of the low-order half, CF and OF are 0; otherwise, they are 1.

Here are some multiplication examples:

```
MUL    BX          ;Unsigned multiply of BX times AX.
MUL    MEM_BYTE    ;Unsigned multiply of memory times AL.
IMUL   DL          ;Signed multiply of DL times AL.
IMUL   MEM_WORD    ;Signed multiply of memory times AX.
```

Neither MUL nor IMUL lets you multiply by an immediate value directly; you must put it in a register or a memory location. For example,

```
MOV   DX,10
MUL   DX
```

multiplies AX by 10.

ASCII Adjust for Multiplication (AAM)

The AAM instruction converts the product of a preceding byte multiplication into two valid unpacked decimal operands. It assumes that the double-length product is in AH and AL, and returns the unpacked operands in AH and AL. For AAM to work correctly, the original multiplier and multiplicand must have been valid unpacked bytes.

To make the conversion, AAM divides the AL register by 10 and stores the resulting quotient and remainder in AH and AL, respectively. It also updates the Parity Flag (PF), the Zero Flag (ZF), and the Sign Flag (SF) to reflect the contents of AL. The Carry Flag (CF), Auxiliary Carry Flag (AF), and Overflow Flag (OF) are undefined.

To see how AAM works, assume AL contains 9 (00001001B) and BL contains 7 (00000111B). The instruction

```
MUL   BL
```

multiplies AL by BL and returns a 16-bit result in AH and AL. In this case, it returns 0 in AH and 00111111B (decimal 63) in AL.

The subsequent instruction

```
AAM
```

divides AL by 10 and returns a quotient of 00000110B in AH and a remainder of 00000011B in AL. This double-length result is indeed correct—BCD 63, in unpacked form.

The 8088 has no instruction that multiplies *packed* decimal numbers. If you want to do this, you must unpack them (with AAM), multiply, then pack the result.

Division Instructions

Divide, Unsigned (DIV) and Integer Divide, Signed (IDIV)

Just as the 8088 has two separate multiplication instructions, it also has two separate division instructions. Divide (DIV) performs an unsigned division and Integer Divide (IDIV) performs a signed division.

These instructions have the general forms

```
DIV   source
IDIV  source
```

where *source* is a byte- or word-length divisor in a general register or a memory location. The dividend is a double-length operand in either AH and AL (8-bit operation) or DX and AX (16-bit operation). The results are as follows:

- If the source operand is a *byte,* the quotient is returned in AL and the remainder in AH.
- If the source operand is a *word,* the quotient is returned in AX and the remainder in DX.

Both instructions leave the flags undefined, but if the quotient cannot fit in the destination register (AL or AX), the 8088 has a dramatic way of telling you the result is invalid: it generates *a type 0 (divide by 0) interrupt.*

The following conditions cause a divide overflow:

1. The divisor is zero.
2. For an unsigned byte divide, the dividend is at least 256 times larger than the divisor.
3. For an unsigned word divide, the dividend is at least 65,536 times larger than the divisor.
4. For a signed byte divide, the quotient exceeds +127 or −128.
5. For a signed word divide, the quotient exceeds +32,767 or −32,768.

Here are some typical division operations:

```
DIV   BX          ;Divide DX:AX by BX, unsigned.
DIV   MEM_BYTE    ;Divide AH:AL by memory, unsigned.
IDIV  DL          ;Divide AH:AL by DL, signed.
IDIV  MEM_WORD    ;Divide DX:AX by memory, signed.
```

Neither DIV nor IDIV lets you divide by an immediate value directly; you must put it in a register or memory location. For instance,

```
MOV  BX,20
DIV  BX
```

divides DX:AX by 20.

ASCII Adjust for Division (AAD)

The decimal adjust instructions we previously described (AAA, DAA, AAS, DAS, and AAM) all operate on the *result* of an operation. By contrast, you must apply the AAD instruction *before* you execute a division operation.

AAD converts an unpacked dividend to a binary value in AL. In doing this, it multiplies the high-order digit of the dividend (the contents of AH) by 10 and adds the result to the low-order digit in AL. Then it zeros the contents of AH.

This sequence shows a typical use of AAD:

```
AAD          ;Adjust the unpacked dividend in AH:AL
DIV  BL      ; then perform the division.
```

Sign-Extension Instructions

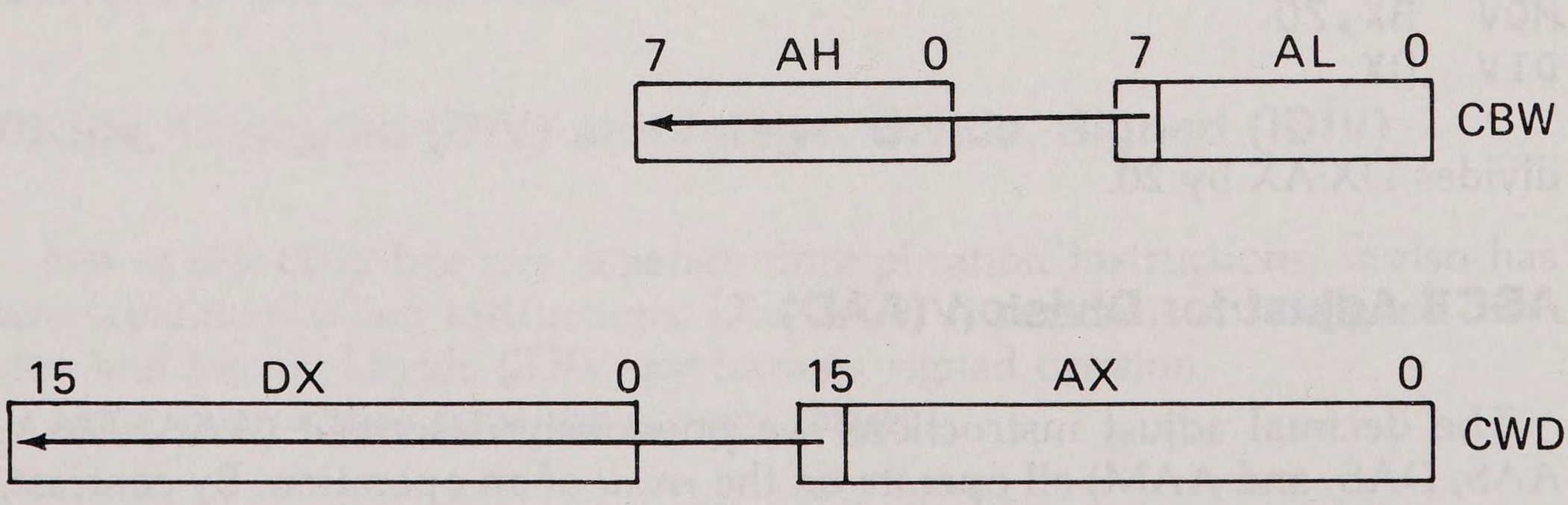
Two instructions let you operate on mixed-size data, by doubling the length of a signed operand. Convert Byte to Word (CBW) reproduces Bit 7 of AL throughout AH. Convert Word to Doubleword (CWD) reproduces Bit 15 of AX throughout DX. Figure 3-7 illustrates these operations.

Thus, CBW lets you add a byte to a word, subtract a word from a byte, and so forth. Similarly, CWD lets you divide a word by a word. Here are some examples:

```
CBW          ;Add a byte in AL to a word in BX.
ADD   AX,BX

CBW          ;Multiply a byte in AL by a word in BX.
IMUL  BX

CWD          ;Divide a word in AX by a word in BX.
IDIV  BX
```



3-7. How CBW and CWD sign-extend data.

3.6 Bit Manipulation Instructions

These instructions manipulate bit patterns within registers and memory locations. Table 3-6 divides them into three groups: logical, shift, and rotate.

Table 3-6. Bit manipulation instructions.

			Flags								
Mnemonic	*Assembler Format*		*OF*	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
Logical											
AND	AND	destination,source	0	-	-	-	*	*	?	*	0
OR	OR	destination,source	0	-	-	-	*	*	?	*	0
XOR	XOR	destination,source	0	-	-	-	*	*	?	*	0
NOT	NOT	destination	-	-	-	-	-	-	-	-	-
TEST	TEST	destination,source	0	-	-	-	*	*	?	*	0
Shift											
SAL/SHL	SAL	destination,count	*	-	-	-	*	*	?	*	*
SAR	SAR	destination,count	*	-	-	-	*	*	?	*	*
SHR	SHR	destination,count	*	-	-	-	0	*	?	*	*
Rotate											
ROL	ROL	destination,count	*	-	-	-	-	-	-	-	*
ROR	ROR	destination,count	*	-	-	-	-	-	-	-	*
RCL	RCL	destination,count	*	-	-	-	-	-	-	-	*
RCR	RCR	destination,count	*	-	-	-	-	-	-	-	*

Note: * means changed, - means unchanged, and ? means undefined.

Logical Instructions

Logical instructions are so named because they operate according to the rules of formal logic, rather than those of mathematics. For example, the rule of logic stated

If A is true and B is true, then C is true

has an 8088 counterpart in the AND instruction, which applies this rule to corresponding bits in two operands.

Specifically, for each bit position where both operands are 1 (true), AND sets the corresponding bit in the destination operand to 1. Conversely, for any bit position where the two operands have any other combination—both

are 0 or one is 0 and the other is 1—AND sets the bit in the destination to 0.

Since logical operations reference bits within an operand, you usually use hexadecimal numbering for the operands. The 8088's logical instructions can operate on either bytes or words, so you normally deal with either two or four hexadecimal digits.

To help you construct the correct "mask" value for a logical operation, Table 3-7 shows the hexadecimal representation of a 1 in 16 different bit positions. For example, to operate on bit 2, the correct mask value is 4H; to operate on bits 2 and 3, the mask is 0CH (hex 4 + hex 8); and so on.

Table 3-7. Hexadecimal values for bit positions.

Bit Number	*Hex Value*	*Bit Number*	*Hex Value*
0	0001	8	0100
1	0002	9	0200
2	0004	10	0400
3	0008	11	0800
4	0010	12	1000
5	0020	13	2000
6	0040	14	4000
7	0080	15	8000

Logical AND (AND), Inclusive-OR (OR), and Exclusive-OR (XOR)

These names should look familiar, because you encountered AND, OR, and XOR in Section 2.7, where we discussed logical operators with the same names. However, operators do their work when the program is assembled, while instructions do theirs when the program is executed. Still, these operators and instructions have the same names because they function in the same way. Although we describe the AND, OR, and XOR instructions here for the sake of completeness, you might want to refer to Section 2.7 for explanations of how they operate.

Table 3-8. The AND, OR, and XOR bit combinations.

		Result		
Source	*Destination*	*AND*	*OR*	*XOR*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

AND, OR, and XOR operate on byte or word operands. They let you combine two registers, a register with a memory location, or an immediate value with a register or memory location. Table 3-8 shows what effect these instructions have.

The *AND* instruction masks out (zeros) certain bits so you can do some kind of processing on the remaining bits. As we just mentioned, for each bit position in which both operands contain 1, the corresponding bit in the destination is also 1. All other operand bit combinations put a 0 in the destination bit. Note that *any bit ANDed with 0 becomes 0, and any bit ANDed with 1 retains its original value.*

Some examples of AND are:

```
AND   AX,BX                ;AND two registers.
AND   AL,MEM_BYTE          ;AND register with memory
AND   MEM_BYTE,AL          ; or vice versa.
AND   BL,1101B             ;AND a constant with a register
AND   TABLE[BX],MASK3      ; or with memory.
```

To see how AND works, suppose port 200 is connected to the 16-bit status register of an external device in the system, and bit 6 indicates whether that device's power is on (1) or off (0). If your program requires device power to be on before continuing, it might include the following loop:

```
CHK_PWR: IN    AX,200          ;Read device status.
         AND   AX,1000000B     ;Isolate the power indicator.
         JZ    CHK_PWR         ;Wait until power is on,
         ..                    ; then continue.
         ..
```

The JZ (Jump If Zero) instruction, which we have not yet discussed, makes the 8088 jump back to the IN instruction at CHK_PWR if the Zero Flag (ZF) is 1, or continue to the next instruction otherwise. Here, ZF is 1 only when the power indicator—bit 6—is 1, because the AND instruction already zeroed the other bits in AX.

The *OR* instruction produces a 1 result in the destination for each bit position in which either or both operands contain 1. OR is generally used to force specific bits to 1. For example,

```
OR   BX,0C000H
```

sets the two most-significant bits (14 and 15) of BX to 1 and leaves all other bits unchanged.

The *XOR* instruction is used to determine which bits differ between two operands, or to reverse the state of selected bits. XOR produces a 1 in the destination for every bit position in which the operands differ; that is, for

every bit position in which one operand has 0 and the other has 1. If both operands contain either 0 or 1, XOR clears the destination bit to 0.

For example,

```
XOR  BX,0C000H
```

reverses the state of the two most-significant bits of BX (14 and 15) and leaves all other bits unchanged.

Logical NOT (NOT)

The NOT instruction reverses the state of each bit in a register or memory operand, without affecting any flags. That is, NOT changes each 1 to 0 and each 0 to 1. In other words, it takes the *one's-complement* of an operand.

Test (TEST)

The TEST instruction logically ANDs two operands, but affects only the flags; it doesn't alter either operand. TEST affects the flags the same as AND: it clears CF and OF to 0, updates PF, ZF, and SF, and leaves AF undefined.

If you follow TEST with a JNZ (Jump If Not Zero) instruction, the jump will be taken if there are any corresponding 1 bits in both operands.

Shift and Rotate Instructions

The 8088 has seven instructions that displace the 8- or 16-bit contents of a general register or memory location one or more positions to the left or right. Three of these instructions "shift" the operand, the other four "rotate" it.

For all seven instructions, the Carry Flag (CF) acts as a "9th bit" or "17th bit" extension of the operand. That is, CF receives the value of the bit that has been displaced out of one end of the operand. A right shift or rotate puts the value of bit 0 into CF. A left shift or rotate puts the value of bit 7 (of a byte) or bit 15 (of a word) into it.

Shift and rotate instructions fall into two groups. *Logical* instructions displace an operand without regard to its sign; they are used to operate on unsigned numbers and non-numbers such as masks. *Arithmetic* instructions preserve the most-significant bit of the operand, the sign bit; they are used to operate on signed numbers. Figure 3-8 shows how these instructions work.

The shift and rotate instructions take two operands: a *destination* and a *count.* The destination may be an 8- or 16-bit general register or memory location. The count may be "1" or an unsigned value in the CL register.

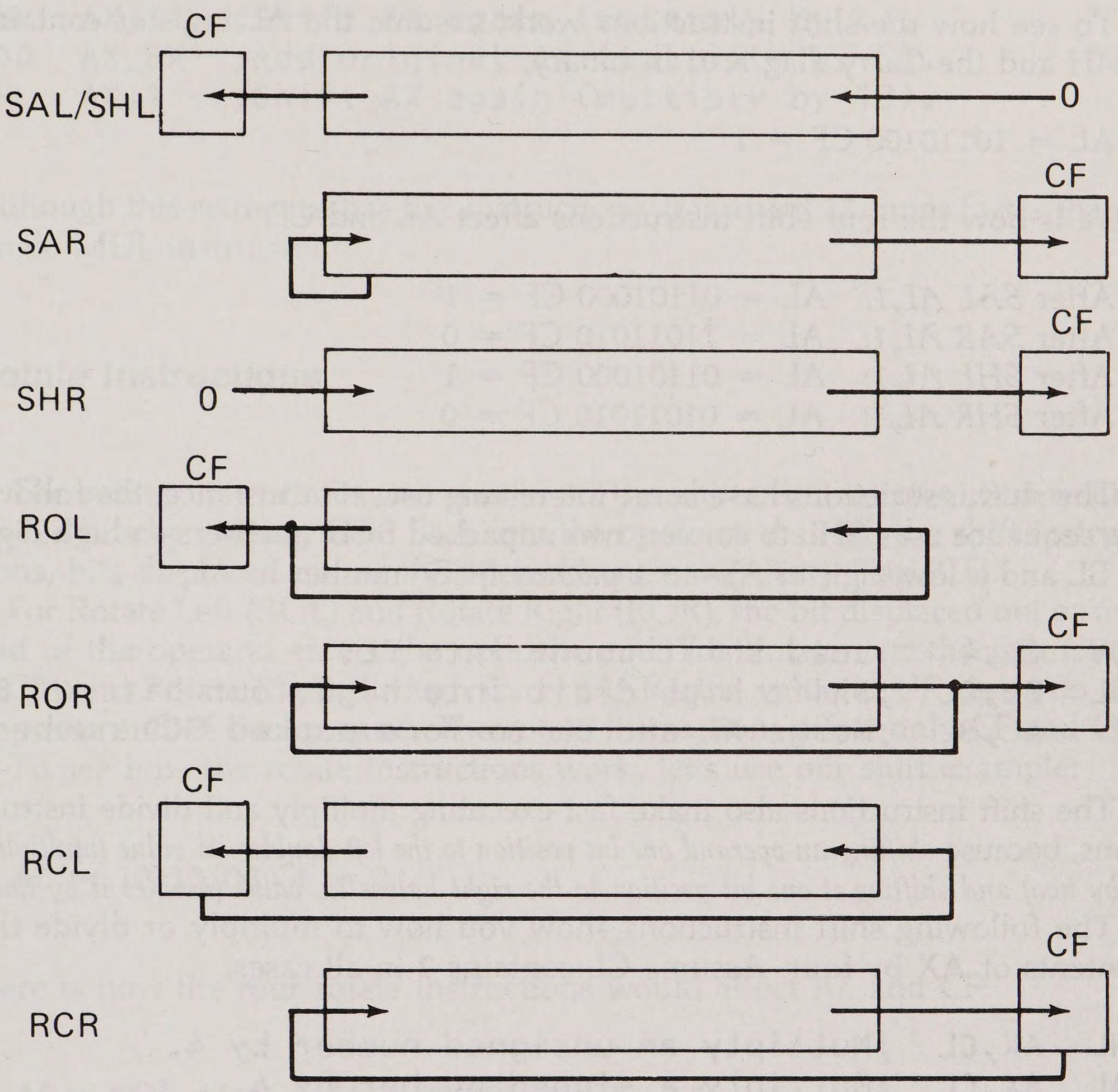


Figure 3-8. Shift and rotate operations.

Shift Instructions

Shift Arithmetic Left (SAL) and *Shift Arithmetic Right (SAR)* shift signed numbers. SAR preserves the sign of the operand by replicating the sign bit throughout the shift operation. SAL does not preserve the sign, but it puts 1 in the Overflow Flag (OF) if the sign ever changes. Whenever SAL shifts an operand, the vacated bit 0 position receives 0.

Shift Logical Left (SHL) and *Shift Logical Right (SHR)* shift unsigned numbers. SHL does the same thing as SAL. SHR is similar to SHL, but it shifts operands right instead of left. Whenever SHR shifts an operand, the vacated high-order bit position (bit 7 in a byte, bit 15 in a word) receives 0.

Besides CF and OF, the shift instructions update PF, ZF, and SF, and leave AF undefined.

To see how the shift instructions work, assume the AL register contains 0B4H and the Carry Flag is 1. In binary,

AL = 10110100 CF = 1

Here is how the four shift instructions affect AL and CF:

After *SAL AL,1:* AL = 01101000 CF = 1
After *SAR AL,1:* AL = 11011010 CF = 0
After *SHL AL,1:* AL = 01101000 CF = 1
After *SHR AL,1:* AL = 01011010 CF = 0

The shift instructions have some interesting uses. For instance, the following sequence uses SHL to convert two unpacked BCD numbers—a high digit in BL and a low digit in AL—to a packed BCD number in AL:

```
MOV   CL,4    ;Load shift count into CL.
SHL   BL,CL   ;Shift high digit into high four bits of BL.
OR    AL,BL   ;Merge AL and BL to form packed BCD number.
```

The shift instructions also make fast-executing multiply and divide instructions, because *shifting an operand one bit position to the left doubles its value (multiplies it by two) and shifting it one bit position to the right halves its value (divides it by two).*

The following shift instructions show you how to multiply or divide the contents of AX by four. Assume CL contains 2 in all cases.

```
SHL   AX,CL   ;Multiply an unsigned number by 4.
SAL   AX,CL   ;Multiply a signed number by 4.
SHR   AX,CL   ;Divide an unsigned number by 4.
SAR   AX,CL   ;Divide a signed number by 4.
```

You save considerable time by multiplying and dividing with shift instructions instead of with the 8088's multiply and divide instructions. Each of the preceding shift instructions takes 16 clock cycles to execute. It takes four more cycles to initialize the CL register, for a total of 20 cycles. Compare this with the minimum execution times for MUL (118 cycles), IMUL (128 cycles), DIV (144 cycles), and IDIV (165 cycles) and you will see that the shift instructions make these calculations about *six to eight times faster* than their multiply and divide counterparts!

While a single shift instruction can multiply or divide only by multiples of two, you can multiply or divide by other factors by juggling some registers. For example, this sequence multiplies AX by 10:

```
MOV   BX,AX   ;Save contents of AX in BX.
SHL   AX,1    ;Shift AX (multiply by 2).
```

```
SHL   AX,1    ;Shift AX again (multiply by 4).
ADD   AX,BX   ;Add original AX (multiply by 5).
SHL   AX,1    ;Shift AX again (multiply by 10).
```

Although this sequence has five instructions, it is about 11 times faster than a single MUL instruction!

Rotate Instructions

The rotate instructions are similar to the shifts, but rotates *preserve* displaced bits by putting them back into the operand. As with the shift instructions, bits displaced out of the operand enter the Carry Flag (CF).

For Rotate Left (ROL) and Rotate Right (ROR), the bit displaced out of one end of the operand enters the opposite end. For Rotate Left through Carry (RCL) and Rotate Right through Carry (RCR), the value of *CF* goes into the opposite end of the operand. The rotate instructions affect only CF and OF.

To see how the rotate instructions work, let's use our shift example:

AL = 10110100 CF = 1

Here is how the four rotate instructions would affect AL and CF:

After *ROL AL,1:* AL = 01101001 CF = 1
After *ROR AL,1:* AL = 01011010 CF = 0
After *RCL AL,1:* AL = 01101001 CF = 1
After *RCR AL,1:* AL = 11011010 CF = 0

3.7 Control Transfer Instructions

As we mentioned earlier, instructions are stored consecutively in memory, but programs rarely execute in that exact order. All but the simplest programs include jumps and procedure calls that alter the execution path the microprocessor takes.

The control transfer instructions can transfer program execution from one part of memory to another. As Table 3-9 shows, these instructions can be divided into three groups: unconditional transfer, conditional transfer, and iteration control. Note that none of them affects the flags.

Table 3-9. Control transfer instructions.

			Flags								
Mnemonic	*Assembler Format*		*OF*	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
Unconditional Transfers											
CALL	CALL	target	-	-	-	-	-	-	-	-	-
RET	RET	[pop-value]	-	-	-	-	-	-	-	-	-
JMP	JMP	target	-	-	-	-	-	-	-	-	-
Conditional Transfers											
JA/JNBE	JA	short-label	-	-	-	-	-	-	-	-	-
JAE/JNB	JAE	short-label	-	-	-	-	-	-	-	-	-
JB/JNAE/JC	JB	short-label	-	-	-	-	-	-	-	-	-
JBE/JNA	JBE	short-label	-	-	-	-	-	-	-	-	-
JCXZ	JCXZ	short-label	-	-	-	-	-	-	-	-	-
JE/JZ	JE	short-label	-	-	-	-	-	-	-	-	-
JG/JNLE	JG	short-label	-	-	-	-	-	-	-	-	-
JGE/JNL	JGE	short-label	-	-	-	-	-	-	-	-	-
JL/JNGE	JL	short-label	-	-	-	-	-	-	-	-	-
JLE/JNG	JLE	short-label	-	-	-	-	-	-	-	-	-
JNC	JNC	short-label	-	-	-	-	-	-	-	-	-
JNE/JNZ	JNE	short-label	-	-	-	-	-	-	-	-	-
JNO	JNO	short-label	-	-	-	-	-	-	-	-	-
JNP/JPO	JNP	short-label	-	-	-	-	-	-	-	-	-
JNS	JNS	short-label	-	-	-	-	-	-	-	-	-
JO	JO	short-label	-	-	-	-	-	-	-	-	-
JP/JPE	JP	short-label	-	-	-	-	-	-	-	-	-
JS	JS	short-label	-	-	-	-	-	-	-	-	-
Iteration Controls											
LOOP	LOOP	short-label	-	-	-	-	-	-	-	-	-
LOOPE/ LOOPZ	LOOPE	short-label	-	-	-	-	-	-	-	-	-
LOOPNE/ LOOPNZ	LOOPNE	short-label	-	-	-	-	-	-	-	-	-

Note: - means unchanged.

Unconditional Transfer Instructions

Procedures

Up to now, the examples in this book contained instructions that do something *once*. From this you might assume that if you need to perform a specific operation at more than one place in your program, you must duplicate the

entire sequence of instructions at each place you need it. Clearly, duplicating a sequence of instructions at many places in a program is frustrating and time-consuming. It also makes programs much longer than they would be if you could avoid this duplication.

As a matter of fact, you *can* eliminate needless duplicating if you define the repeated instructions as a *procedure.* A procedure (called a subroutine in many instances) is a set of instructions that you write just once, but which you can execute as needed at any point in a program.

The process of transferring control from the main part of a program to a procedure is defined as "calling." Thus, procedures are *called.* Once called, the 8088 executes the instructions in the procedure, then returns to the place where the call was made.

This invites two questions: "How do you call a procedure?" and, "How does the 8088 return to the proper place in the program?" The answers involve two procedure-related instructions, CALL and RET.

Call a Procedure (CALL) and Return from Procedure (RET)

Instructions that execute procedures must perform three tasks:

1. They must include some provision for saving the contents of the Instruction Pointer (IP). Once the procedure has been executed, this address is used to return the 8088 to the calling place. Hence, we call it a *return address.*
2. They must make the microprocessor begin executing the procedure.
3. They must use the stored contents of the IP to return to the program, and continue executing at this point.

These three tasks are performed by two instructions: *Call a Procedure (CALL)* and *Return from Procedure (RET).* They are, essentially, the assembly language equivalents of GOSUB and RETURN in BASIC.

CALL performs the return address-storing and begin-executing tasks (1 and 2). The return address it pushes onto the stack is 16 bits long if you defined the procedure as NEAR or 32 bits if you defined it as FAR (see Section 2.4). NEAR procedures can be called only from within the segment in which they reside; FAR procedures can be called from a different segment.

CALL has the general format

```
CALL  target
```

where *target* is the name of the procedure being called. If *target* is NEAR, CALL pushes the offset address of the next instruction onto the stack. If *target* is FAR, it pushes the contents of the CS register, then the offset address.

After saving the return address, CALL loads the offset address of the target label into IP. If the procedure is FAR, it also loads the label's segment number into CS.

The RET instruction makes the 8088 leave the procedure and return to the calling program. It does this by "undoing" everything CALL did. RET must always be the last procedure instruction that the 8088 executes. (This doesn't mean that RET must be the last instruction in the procedure—although it often is—just the last one the 8088 *executes*.)

RET pops the return address off the stack. If the procedure is NEAR (in the same code segment as the CALL), RET pops one word off the stack, and loads it into the Instruction Pointer (IP). If the procedure is FAR (in a different code segment), RET pops two words off the stack: an offset address for IP, then a segment number for CS.

For example, to call a NEAR procedure named MY_PROC at some point, your program might execute the following sequence (offsets are also listed):

```
04F0             CALL   MY_PROC   ;Call the procedure.
04F3  NEXT       MOV    AX,BX     ;Return here after the procedure.
                 ..
                 ..
0500  MY_PROC    PROC             (start of procedure)
0500             MOV    CL,6      ;First instruction of procedure.
                 ..               (additional instructions)
                 ..
051E             RET              ;Return to calling program.
                 ..
051F  MY_PROC    ENDP              (end of procedure)
```

When the 8088 executes CALL, it pushes the offset address of NEXT (04F3H) onto the stack, then loads the offset address of MY_PROC (0500H) into the Instruction Pointer (IP). Since the PROC directive has no distance operand, MY_PROC is (by default) a NEAR procedure.

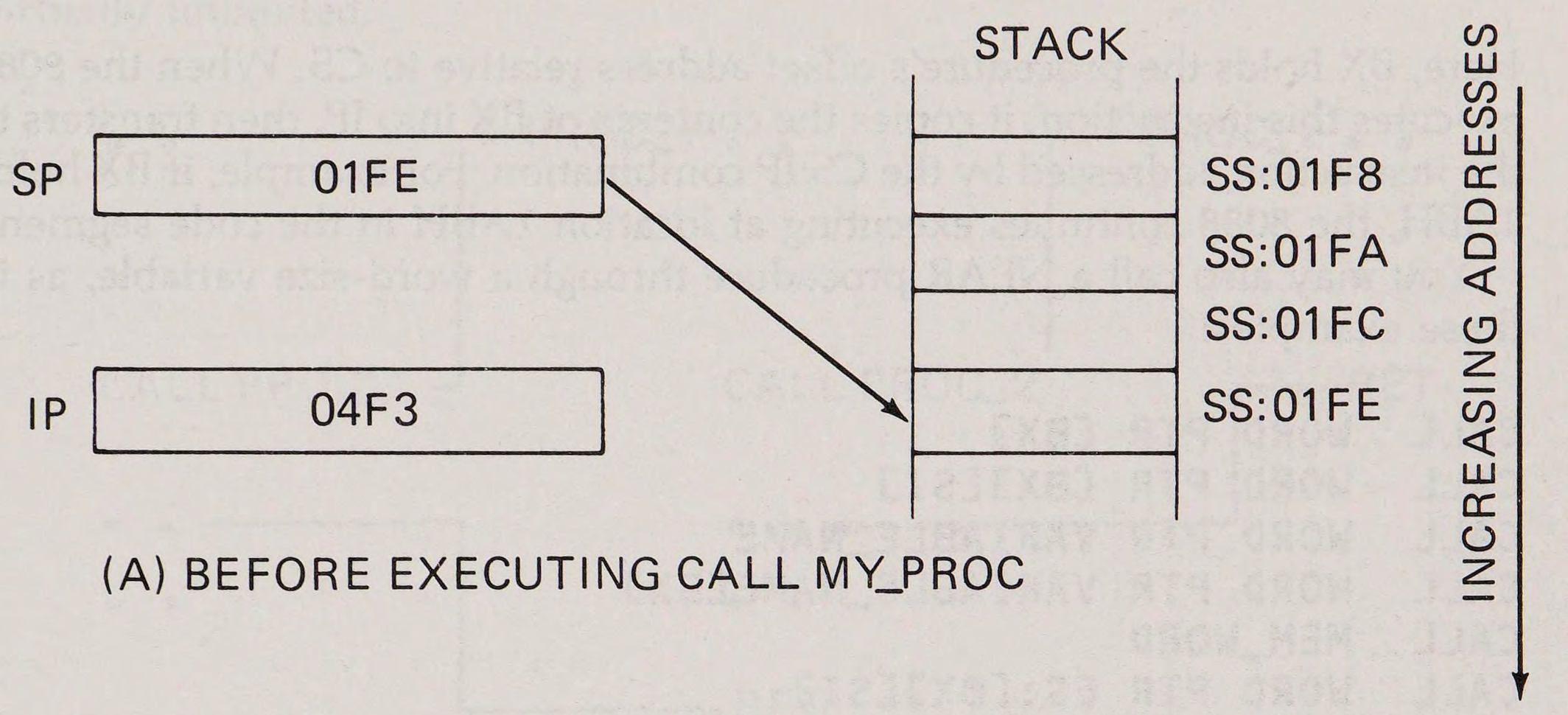
Since IP has changed, the 8088 begins executing at that new offset address. In our example, the first instruction happens to be

```
MOV CL,6
```

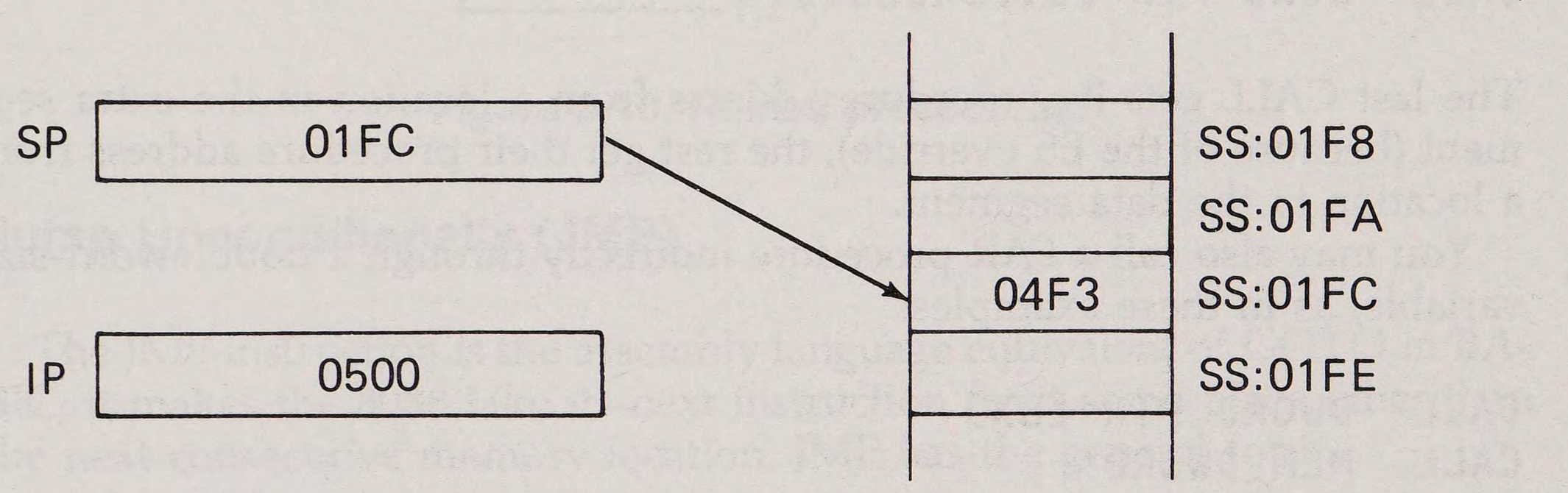
When the microprocessor encounters the RET instruction, it pops the return address off the stack and puts it in IP. This makes it resume at the instruction labeled NEXT. Figure 3-9 shows the stack, the Stack Pointer (SP), and the Instruction Pointer (IP) before and after the CALL, and after RET.

Indirect Calls to Procedures

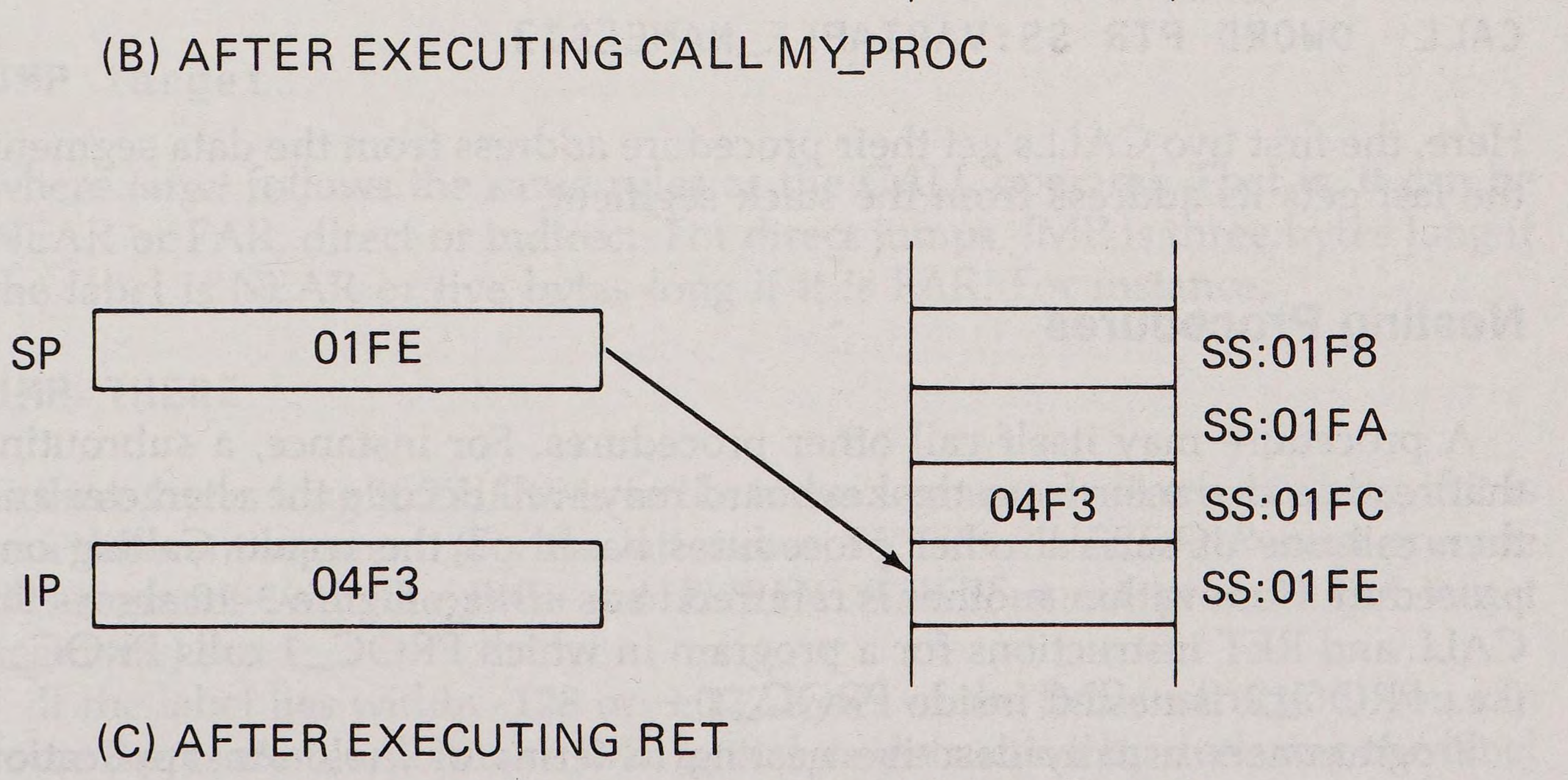
So far we have discussed just one form of the CALL instruction: a *direct* call, in which the operand is a label. You may also make an *indirect* call to a procedure, through a register or a memory location. With indirect calls



(A) BEFORE EXECUTING CALL MY_PROC



(B) AFTER EXECUTING CALL MY_PROC



(C) AFTER EXECUTING RET

Figure 3-9. How a procedure affects the stack.

through memory, the 8088 fetches the procedure's IP value from the data segment, unless you use BP or specify an override. If you use BP to address memory, the 8088 fetches the IP value from the stack segment.

You may call a NEAR procedure through a register, like this:

```
CALL BX
```

Here, BX holds the procedure's offset address relative to CS. When the 8088 executes this instruction, it copies the contents of BX into IP, then transfers to the instruction addressed by the CS:IP combination. For example, if BX holds 1ABH, the 8088 continues executing at location 1ABH in the code segment.

You may also call a NEAR procedure through a word-size variable, as in these examples:

```
CALL   WORD PTR [BX]
CALL   WORD PTR [BX][SI]
CALL   WORD PTR VARIABLE_NAME
CALL   WORD PTR VARIABLE_NAME[BX]
CALL   MEM_WORD
CALL   WORD PTR ES:[BX][SI]
```

The last CALL gets its procedure address from a location in the extra segment (because of the ES override); the rest get their procedure address from a location in the data segment.

You may also call a FAR procedure indirectly through a doubleword-size variable, as in these examples:

```
CALL   DWORD PTR [BX]
CALL   MEM_DWORD
CALL   DWORD PTR SS:VARIABLE_NAME[SI]
```

Here, the first two CALLs get their procedure address from the data segment; the last gets its address from the stack segment.

Nesting Procedures

A procedure may itself call other procedures. For instance, a subroutine that reads a character from the keyboard may well decode the character and then call one of several other procedures based on the result. Calling one procedure from within another is referred to as *nesting*. Figure 3-10 shows the CALL and RET instructions for a program in which PROC_1 calls PROC_1 (i.e., PROC_2 is nested inside PROC_1).

Programmers usually describe nesting in terms of *levels*. An application like the one in Figure 3-10, where the nesting extends only to the CALL to

PROC_2 (PROC_2 does not call another procedure), is said to have one level of nesting. However, PROC_2 might well have called a third procedure (PROC_3), with PROC_3 calling PROC_4, and so on.

Because each CALL instruction pushes two or four address bytes onto the stack, nesting is limited only by the capacity of the stack segment. Since a stack segment can be up to 64K bytes long, your nesting capabilities are virtually unlimited.

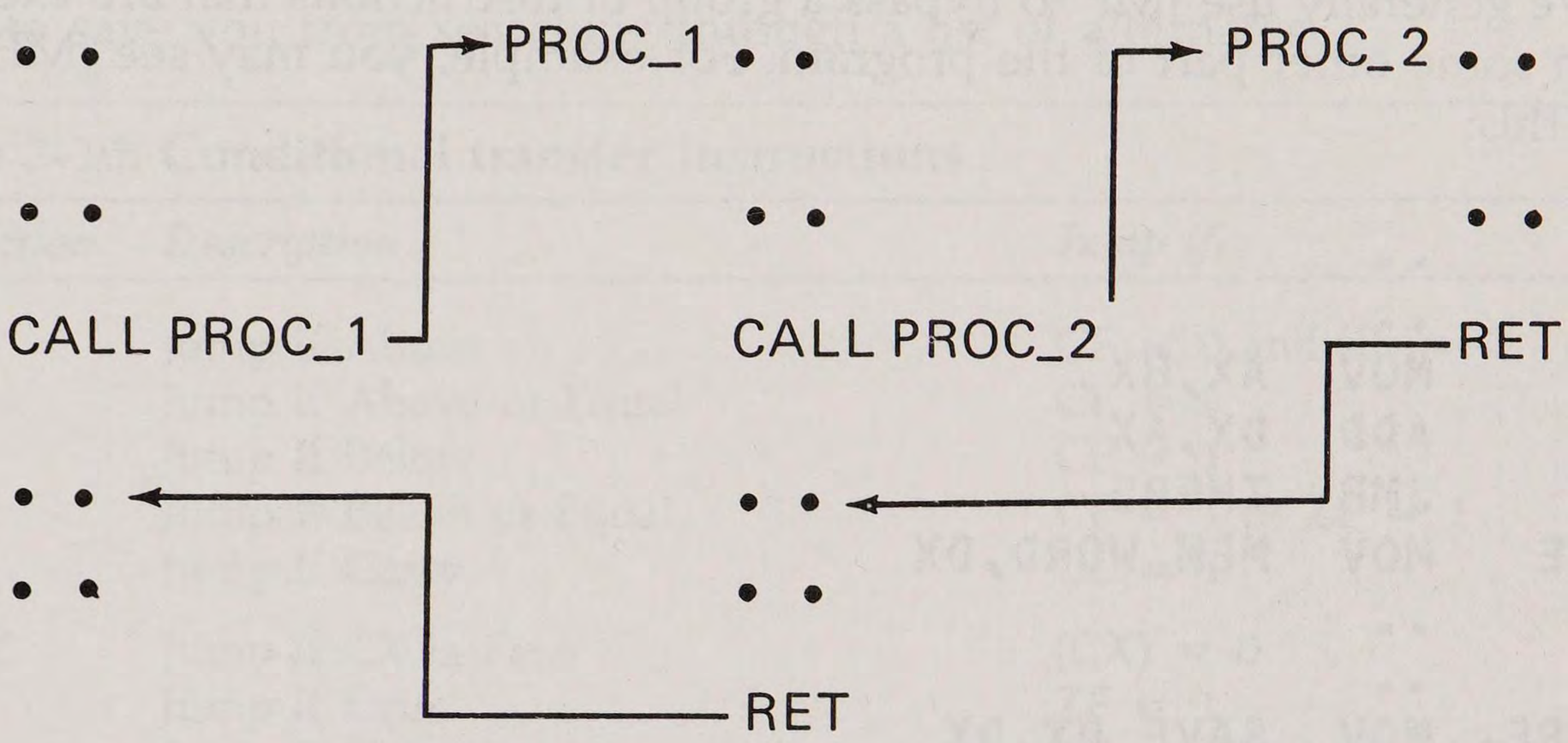


Figure 3-10. Nested procedures.

Jump Unconditionally (JMP)

The JMP instruction is the assembly language equivalent of GOTO in BASIC; it makes the 8088 take its next instruction from some place other than the next consecutive memory location. JMP has the general form

```
JMP target
```

where *target* follows the same rules as the CALL operand. That is, it can be NEAR or FAR, direct or indirect. For direct jumps, JMP is three bytes long if the label is NEAR or five bytes long if it is FAR. For instance,

```
JMP THERE
```

is three bytes long if *THERE* is in the same segment or five bytes long if it is in a different segment. (For the latter case, *EXTRN THERE:FAR* must precede the segment with the JMP and *PUBLIC THERE* must precede the target segment.)

If the label lies within -128 or +127 bytes of the JMP instruction, you can force the assembler to make JMP a *two-byte* instruction by declaring the label SHORT. For example,

```
JMP   SHORT NEAR_LABEL
```

is a two-byte instruction. It executes in the same amount of time as

```
JMP   NEAR_LABEL
```

but occupies one less byte in memory.

We generally use JMP to bypass a group of instructions that are executed from some other part of the program. For example, you may see JMP used like this:

```
          ..
          ..
          MOV   AX,BX
          ADD   DX,AX
          JMP   THERE
HERE      MOV   MEM_WORD,DX
          ..
          ..
THERE     MOV   SAVE_DX,DX
          ..
```

Conditional Transfer Instructions

There are 17 different instructions that let the 8088 make an execution "decision" based on some prescribed condition, such as a register value being zero or CF being set to 1. If the condition is satisfied, the 8088 makes the jump; otherwise, it continues to the next instruction in the program.

As shown in Table 3-9, the assembler recognizes some of these instructions by two or three different mnemonics. (With these alternate mnemonics, one might well claim that the 8088 has *31* conditional transfer instructions. If you think that way, you should seriously consider a career in marketing!)

For example, the assembler treats JA LABEL and JNBE LABEL as the same instruction. These particular instructions refer to the result of a preceding Compare (CMP) or Subtract (SUB or SBB) instruction.

The first mnemonic, JA, tells the 8088 to make the jump if the destination is "Above" (that is, greater than) the source. The second, JNBE, tells it to make the jump if the destination is "Not Below Nor Equal" to the source. Thus, JA and JNBE say exactly the same thing, but in different ways. They exist strictly for your convenience, so you can write programs that are more readily understandable.

The conditional transfer instructions have the general format

```
Jx  short-label
```

where *x* is a one- to three-letter modifier. The operand form *short-label* tells you that the target label of the jump must be no more than −128 or +127 bytes away from the conditional transfer instruction. Contrast this with the Jump (JMP) instruction, which can transfer anywhere in memory.

Table 3-10 summarizes the conditional transfer instructions, and shows which conditions cause a jump. Here, we have listed the mnemonics individually to save you from searching through a list of alternates.

Table 3-10. Conditional transfer instructions.

Instruction	*Description*	*Jump if. . .*
JA	Jump If Above	CF = 0 and ZF = 0
JAE	Jump If Above or Equal	CF = 0
JB	Jump If Below	CF = 1
JBE	Jump If Below or Equal	CF = 1 or ZF = 1
JC	Jump If Carry	CF = 1
JCXZ	Jump If CX is Zero	(CX) = 0
JE	Jump If Equal	ZF = 1
*JG	Jump If Greater	ZF = 0 and SF = OF
*JGE	Jump If Greater or Equal	SF = OF
*JL	Jump If Less	SF not = OF
*JLE	Jump If Less or Equal	ZF = 1 or SF not = OF
JNA	Jump If Not Above	CF = 1 or ZF = 1
JNAE	Jump If Not Above nor Equal	CF = 1
JNB	Jump If Not Below	CF = 0
JNBE	Jump If Not Below nor Equal	CF = 0 and ZF = 0
JNC	Jump If No Carry	CF = 0
JNE	Jump If Not Equal	ZF = 0
*JNG	Jump If Not Greater	ZF = 1 or SF not = OF
*JNGE	Jump If Not Greater nor Equal	SF not = OF
*JNL	Jump If Not Less	SF = 0F
*JNLE	Jump If Not Less Nor Equal	ZF = 0 and SF = OF
*JNO	Jump If No Overflow	OF = 0
JNP	Jump If No Parity (Odd)	PF = 0
*JNS	Jump If No Sign	SF = 0
JNZ	Jump If Not Zero	ZF = 0
*JO	Jump On Overflow	OF = 1
JP	Jump On Parity (Even)	PF = 1
JPE	Jump If Parity Even	PF = 1
JPO	Jump If Parity Odd	PF = 0
*JS	Jump On Sign	SF = 1
JZ	Jump If Zero	ZF = 1

*Pertinent for signed (two's-complement) arithmetic.

Conditional transfer instructions occupy two bytes in memory; the first holds the operation code, the second holds the relative displacement. These instructions execute in 16 clock cycles if the jump is taken, and four clock cycles if it is not taken. Because of this time difference, you should construct your programs so that, whenever possible, the expected case executes if the jump is *not taken.*

Here are some examples of conditional transfer instructions:

1. The sequence

```
ADD   AL,BL
JC    TOOBIG
```

   jumps to TOOBIG if the addition produces a carry.
2. The sequence

```
SUB   AL,BL
JZ    ZERO
```

   jumps to ZERO if the subtraction produces a zero result in AL.
3. The sequence

```
CMP   AL,BL
JE    ZERO
```

   jumps to ZERO if AL and BL hold the same value. (We could have used the equivalent mnemonic JZ here, but JE—for "Jump if Equal"—is more meaningful in this case.)
4. Some tests require you to choose between two different conditional transfer instructions, based on whether you are testing the result of a signed or unsigned operation. For example, suppose you want to jump to label BXMORE if the contents of BX are higher-valued than those of AX.

   The sequence

```
CMP   BX,AX
JA    BXMORE
```

   applies if the operands are *unsigned,* whereas

```
CMP   BX,AX
JG    BXMORE
```

   applies if the operands are *signed.*

Using Conditional Transfers with Compare (CMP)

You can precede conditional transfer instructions with any instruction that alters the flags, but you normally precede them with a Compare (CMP) instruction. Table 3-5 in Section 3.5 shows how CMP affects the flags for various source and destination relationships. Now, with the wide variety of conditional transfer instructions at your disposal, it is worthwhile to look at a more practical table—one that shows which conditional transfer to use for all possible combinations of source and destination. Table 3-11 is the one you need.

Table 3-11. Using conditional transfers with Compare (CMP).

	Follow CMP with	
To Jump If	*For unsigned numbers*	*For signed numbers*
Destination is greater than Source	JA	JG
Destination is equal to Source	JE	JE
Destination is not equal to Source	JNE	JNE
Destination is less than Source	JB	JL
Destination is less than or equal to Source	JBE	JLE
Destination is greater than or equal to Source	JAE	JGE

To illustrate a typical application for a conditional transfer/compare combination, Example 3-1 shows a program sequence that arranges two unsigned numbers in memory in increasing order. The data segment offsets of these numbers are assumed to be in BX and DI, respectively. Note that whenever two numbers need to be exchanged, one of them must be loaded into a register, because the 8088 has no memory-to-memory move instruction.

You can also combine a single compare instruction with two conditional transfer instructions, to test the "less than," "equal to," and "greater than" cases separately. Example 3-2 shows a sequence that executes any of three groups of instructions, based on whether the value in AL is below, equal to, or above 10.

This sequence uses a JAE instruction to determine whether AL is above or equal to 10. If it is, the 8088 jumps to the label AE10. A JA instruction then determines whether AL is above 10. If it is, the 8088 jumps to A10. Normally, the last instruction in the first two groups would be a JMP, to skip the unused options.

Example 3-1. Arranging two numbers in increasing order.

```
;  This sequence arranges two unsigned 16-bit numbers in memory
;  in order of value, with the lower-numbered value in the lower
```

```
;  address. The offset of the first number is in BX; the offset
;  of the second number is in DI.

        MOV   AX,[BX]  ;Load first number into AX.
        CMP   AX,[DI]  ;Compare it with second number.
        JBE   DONE     ;First no. below or equal to second?
        XCHG  AX,[DI]  ; If not, swap the two numbers.
        MOV   [BX],AX
DONE:   ..
        ..
```

Example 3-2. Three-way decision sequence.

```
;  This sequence executes one of three different groups of
;  instructions, based on whether the unsigned number in
;  AL is below, equal to, or above 10.

        CMP  AL,10    ;Compare AL to 10.
        JAE  AE10
        ..            ;Instructions for AL < 10.
        ..
AE10:   JA   A10
        ..            ;Instructions for AL = 10.
        ..
A10:    ..            ;Instructions for AL > 10.
        ..
```

Iteration Control Instructions

The iteration control instructions are conditional transfers used to set up repetitive loops. In the 8088, the Count register (CX) serves as the counter for loops. Each iteration control instruction decrements CX by 1, then makes a jump/no-jump "decision" based on its new contents.

The basic instruction of this group, *Loop until Count Complete (LOOP)*, decrements CX by 1 and transfers control to a *short-label* target operand if CX is not 0. For example, to perform a certain set of operations 100 times, use this kind of sequence:

```
        MOV   CX,100   ;Load repetition count into CX.
START:  ..             (The instructions to be repeated go here)
        ..
        LOOP  START    ;If CX is not zero, jump to START.
        ..             ;Otherwise continue here.
```

LOOP by itself terminates the loop only if CX has been decremented to 0. However, many applications involve loops that must also terminate if something takes place before CX reaches 0. The instructions that provide this alternate escape route are *Loop If Equal (LOOPE)* and *Loop If Not Equal (LOOPNE)*.

LOOPE, which has the alternate form *Loop If Zero (LOOPZ)*, decrements CX by 1, then jumps if CX is not 0 and the Zero Flag (ZF) is 1. Thus, looping continues until CX is 0 or ZF is 0, or both. You normally use LOOPE to find the first non-zero result in a series of operations. To illustrate, Example 3-3 shows a sequence that finds the first non-zero byte in a block of memory. The offset addresses of the first and last bytes are in BX and DI, respectively.

LOOPNE, which has the alternate form *Loop If Not Zero (LOOPNZ)*, decrements CX by 1, then jumps if CX is not zero and the Zero Flag (ZF) is 0. Thus, looping continues until CX is zero or ZF is 1, or both. LOOPNE is normally used to find the first zero result in a series of operations. If you replace LOOPE with LOOPNE in Example 3-3, the sequence will find the first zero byte in a block of memory rather than the first non-zero byte.

Example 3-3. Find a non-zero location in a memory block.

```
;  This sequence finds the first non-zero byte in a specified
;  block of memory. The offset of the starting address is in
;  BX; the offset of the ending address is in DI. The offset
;  of the non-zero byte is returned in BX. If no non-zero
;  bytes are found, BX will hold the same value as DI upon
;  return.
            SUB   DI,BX                 ;Byte count =
            INC   DI                    ; (DI) - (BX) + 1
            MOV   CX,DI                 ;Move byte count into CX.
            DEC   BX
NEXT:       INC   BX                    ;Point to next location
            CMP   BYTE PTR [BX],0       ;and compare it to 0.
            LOOPE  NEXT                 ;Go compare next byte.
            JNZ   NZ_FOUND              ;Non-zero byte found?
            ..                          ; No. (BX) = (DI)
            ..
NZ_FOUND:   ..                          ; Yes. Offset of the
            ..                          ; non-zero entry is in BX.
```

3.8 String Instructions

The string instructions let you operate on blocks of bytes or words in memory. These blocks, or *strings*, may be up to 64K bytes long, and may consist of numeric values (either binary or BCD), alphanumeric values (such as ASCII text characters), or, for that matter, any kind of information that can be stored in memory as binary patterns.

The string instructions provide five basic operations, called *primitives*, that process strings one element (byte or word) at a time. These primitives—move, compare, scan for a value, load, and store—are summarized in Table 3-12.

Note that each primitive has three different instruction forms. The first form takes one or two operands (e.g., MOVS takes two operands), while the other two take no operands (e.g., MOVSB and MOVSW). The 8088 can only execute the no-operand forms. The assembler always converts the more general operand form to one of these two instructions when you assemble the program.

The 8088 assumes that the destination string is in the extra segment and that the source string is in the data segment. The processor addresses the destination string with DI and the source string with SI. For example, MOVSB copies the data segment byte addressed by SI into the extra segment location addressed by DI. Intel probably chose DI and SI here because they are easy-to-remember abbreviations for Destination Index (DI) and Source Index (SI). Clever, no?

Incidentally, although the 8088 assumes the destination string is in the extra segment and the source string is in the data segment, you *can* use other combinations. We will show you how later.

Because the string instructions are designed to operate on a *series* of elements, they automatically update the pointer(s) to address the next element in the string. For example, MOVS increments or decrements its source string pointer (SI) and destination string pointer (DI) after each execution cycle.

The Direction Flag (DF) bit in the 8088's flags register determines whether SI and DI increment or decrement at the end of a string instruction. If DF is 0, SI and DI *increment* after every instruction; if DF is 1, they *decrement*.

For example, MOVS copies an element from the source string to the destination string. If DF is 0, the 8088 *increments* SI and DI after the move, thereby addressing the following element in memory. If DF is 1, it *decrements* SI and DI after the move, thereby addressing the preceding element in memory.

Two instructions let you control the state of DF: *Clear Direction Flag (CLD)* makes it 0 and *Set Direction Flag (STD)* makes it 1. We discuss CLD and STD in Section 3.10. You can make a single string instruction operate on a

Table 3-12. String instructions.

Mnemonic	*Assembler Format*		*Flags* OF	DF	IF	TF	SF	ZF	AF	PF	CF
Repeat Prefixes											
REP	REP		-	-	-	-	-	-	-	-	-
REPE/REPZ	REPE		-	-	-	-	-	-	-	-	-
REPNE/ REPNZ	REPNE		-	-	-	-	-	-	-	-	-
Move											
MOVS	MOVS	dest-string, source-string	-	-	-	-	-	-	-	-	-
MOVSB	MOVSB		-	-	-	-	-	-	-	-	-
MOVSW	MOVSW		-	-	-	-	-	-	-	-	-
Compare											
CMPS	CMPS	dest-string, source-string	*	-	-	-	*	*	*	*	*
CMPSB	CMPSB		*	-	-	-	*	*	*	*	*
CMPSW	CMPSW		*	-	-	-	*	*	*	*	*
Scan											
SCAS	SCAS	dest-string	*	-	-	-	*	*	*	*	*
SCASB	SCASB		*	-	-	-	*	*	*	*	*
SCASW	SCASW		*	-	-	-	*	*	*	*	*
Load and Store											
LODS	LODS	source-string	-	-	-	-	-	-	-	-	-
LODSB	LODSB		-	-	-	-	-	-	-	-	-
LODSW	LODSW		-	-	-	-	-	-	-	-	-
STOS	STOS	dest-string	-	-	-	-	-	-	-	-	-
STOSB	STOSB		-	-	-	-	-	-	-	-	-
STOSW	STOSW		-	-	-	-	-	-	-	-	-

Note: - means unchanged and * means changed.

number of consecutive elements in memory by preceding it with a *repeat prefix* (see Table 3-12). These are not instructions, but one-byte modifiers that make the 8088 hardware repeat a string instruction. With this special hardware, the 8088 processes long strings much faster than it could with a software loop.

Repeat Prefixes

Repeat prefixes make the 8088 repeat a string instruction the number of times specified in the CX register. For example, the sequence

```
        MOV    CX,500
REP     MOVS   DEST,SOURCE
```

makes it execute the MOVS instruction 500 times (we'll discuss MOVS momentarily), and decrement the CX register after each repetition. In effect, the REP prefix says "repeat while not end-of-string"; that is, repeat while CX is not 0.

The remaining repeat prefixes involve the Zero Flag (ZF) in the repeat/exit "decision." Hence, they apply only to the compare string and scan string instructions, which affect ZF. Repeat While Equal (REPE), which has the alternate name Repeat While Zero (REPZ), repeats the instruction as long as ZF is 1 and CX is not 0. If you attach an REPE prefix to a *string compare instruction (CMPS)*, the compare operation repeats until a mismatch occurs. For example,

```
        MOV    CX,100
REPE    CMPS   DEST,SOURCE
```

compares the strings SOURCE and DEST, element by element, until 100 elements have been compared or until the 8088 finds an element in DEST that does not match the corresponding element in SOURCE.

Repeat While Not Equal (REPNE), which has the alternate name Repeat While Not Zero (REPNZ), has the opposite effect of REPE. That is, REPNE causes the prefixed instruction to be repeated while ZF is 0 and CX is not zero. For example, the sequence

```
        MOV    CX,100
REPNE   CMPS   DEST,SOURCE
```

compares SOURCE to DEST until 100 elements have been compared or until the 8088 finds an element in DEST that *matches* the corresponding element in SOURCE.

Move-String Instructions

Move String (MOVS)

MOVS copies a byte or word from one portion of memory to another. It has the general format

```
MOVS  dest-string,source-string
```

Here, *source-string* is a string in the data segment and *dest-string* is a string in the extra segment. As with CMPS, the 8088 uses SI to address the data segment and DI to address the extra segment. So *MOVS copies a byte or word from the data segment to the extra segment.*

You may override the segment assignment for SI, but not for DI. For instance, you might override SI to copy a string from one part of the extra segment to another.

Although MOVS moves only one element, you can move a string of up to 64K bytes (32K words) by applying the REP prefix. Example 3-4 shows a sequence that copies a 100-byte string called SOURCE in the data segment into the extra segment, where it is called DEST. As demonstrated here, every multi-element MOVS operation takes these five steps:

1. Clear DF (with CLD) or set DF (with STD), depending on whether you want the move to progress toward higher or lower addresses in memory, respectively.
2. Load the offset of the source string into SI.
3. Load the offset of the destination string into DI.
4. Load the element count (number of bytes or words to be moved) into CX.
5. Execute the MOVS instruction, with a REP prefix.

Of course, your program must include an *extra segment* that holds space for the destination string and a *data segment* that holds the source string. Generally, you reserve space for the destination string with a directive such as

```
DEST   DB   100 DUP(?)
```

How does the assembler know whether you are moving bytes or words? It determines this based on the *type* of the source and destination labels in the operand field.

If these labels were set up with DB directives, the assembler encodes MOVS as a MOVSB instruction. Conversely, if the labels were set up with DW directives, the assembler encodes MOVS as a MOVSW instruction.

Therefore, if you define SOURCE and DEST with DW directives, the sequence in Example 3-4 copies 100 words instead of 100 bytes.

Example 3-4. Multi-byte move operation.

```
;  This sequence copies 100 bytes from the SOURCE string in
;  the data segment to the DEST string in the extra segment.

        CLD                         ;Set DF = 0, to move forward.
        LEA    SI,SOURCE            ;Load SOURCE offset into SI
        LEA    DI,ES:DEST           ; and DEST offset into DI.
        MOV    CX,100               ;Load element count into CX.
REP     MOVS   DEST,SOURCE          ;Move the bytes.
```

Move Byte String (MOVSB) and Move Word String (MOVSW)

Besides reminding you which strings are involved in the move, of what value are the MOVS instruction's operands? That is, what do they tell the assembler? They tell it simply to translate MOVS into one of the two forms the 8088 can execute: *MOVSB (Move Byte String)* or *MOVSW (Move Word String).* To make this translation, the assembler doesn't need to know *which* strings are affected (that information is in SI and DI), but only what size elements they have.

Since element size is the only thing the assembler finds out from MOVS, there is no reason we can't use the size-specific instructions MOVSB and MOVSW instead of the general instruction MOVS. In fact, MOVSB and MOVSW are preferable to MOVS, because they save the assembler from looking up the size of the operands. As you might expect, these instructions have no operands; their forms are:

```
MOVSB
MOVSW
```

Thus, for example, we could write the instructions in Example 3-4 as follows:

```
        CLD
        LEA     SI,SOURCE
        LEA     DI,ES:DEST
        MOV     CX,100
REP     MOVSB
```

Overriding the Segment Assignments

SI normally addresses the data segment, but you can use a different segment by applying a *segment override* prefix to the source operand. For example,

```
LEA     SI,ES:HERE
LEA     DI,ES:THERE
MOVSB
```

copies a byte from HERE to THERE, where both strings are in the extra segment.

Since you can't override DI, it appears the destination must always be a string in the extra segment. Does this mean you can't copy a string into the data segment? No, you *can* copy a string into the data segment, but it takes some trickery.

To copy a string into the data segment, you must give the extra segment register (ES) the same value as the data segment register (DS). After that, when you execute MOVS, the 8088 will think it is copying a string from the data segment to the extra segment, as usual. But *you* know it is in fact copying a string from one place in the data segment to another!

Example 3-5 shows this trick being used to copy 100 bytes from SOURCE_D to DEST_D, where both strings are in the data segment. Note that except for the first two instructions, this sequence is identical to Example 3-4. You can also apply this technique to the other string instructions we discuss in this section.

Example 3-5. Data segment move operation.

```
;  This sequence copies 100 bytes from string SOURCE_D
;  to string DEST_D, where both strings are in the data
;  segment.

      PUSH  DS                ;Make ES point to data segment.
      POP   ES
      CLD                     ;Set DF = 0, to move forward.
      LEA   SI,SOURCE_D       ;Load SOURCE_D offset into SI
      LEA   DI,DEST_D         ; and DEST_D offset into DI.
      MOV   CX,100            ;Load element count into CX.
REP   MOVSB                   ;Move the bytes.
```

Compare-String Instructions

Compare Strings (CMPS)

Like the Compare (CMP) instruction we discussed in Section 3.5, the *Compare Strings (CMPS)* instruction compares a source operand to a destination operand, and returns the results in the flags. CMPS, like CMP, affects neither operand.

The general form of CMPS is:

```
CMPS  dest-string,source-string
```

where *source-string* is a string in the data segment addressed by SI and *dest-string* is a string in the extra segment addressed by DI. Therefore, CMPS compares an element (byte or word) in the data segment with an element in the extra segment.

Like the CMP instruction, CMPS compares the operands by subtracting them. However, CMP subtracts the source operand from the destination operand, whereas *CMPS subtracts the destination operand from the source operand!* This means the conditional transfer instructions that follow a CMPS instruction must be different from those that follow a CMP. Table 3-13 is the one you need with CMPS.

To compare more than one element, apply a repeat prefix to the CMPS instruction. Here, REP is meaningless, because it returns the flags result of comparing only the two final elements. You must prefix CMPS with either REPE (REPZ) or REPNE (REPNZ).

Table 3-13. Using conditional transfers with CMPS.

	Follow CMP with	
To Jump If	*For unsigned numbers*	*For signed numbers*
Source is greater than Destination	JA	JG
Source is equal to Destination	JE	JE
Source is not equal to Destination	JNE	JNE
Source is less than Destination	JB	JL
Source is less than or equal to Destination	JBE	JLE
Source is greater than or equal to Destination	JAE	JGE

REPE makes the 8088 compare the strings until either (CX) is zero or it finds two mismatched elements. Thus,

```
        CLD
        MOV     CX,100
REPE    CMPS    DEST,SOURCE
```

compares up to 100 elements of SOURCE and DEST in an attempt to find two elements that are not alike.

REPNE makes the 8088 compare the strings until either (CX) is zero or it finds two identical elements. For example,

```
        CLD
        MOV     CX,100
REPNE   CMPS    DEST,SOURCE
```

compares up to 100 elements of SOURCE and DEST in an attempt to find two elements that are alike.

As with MOVS, the Direction Flag (DF) determines whether the operation proceeds forward (DF = 0) or backward (DF = 1) in memory, and SI and DI are updated after each operation.

Checking Results

Since repeated compare operations can terminate in two ways—the contents of CX are 0 or ZF changed to 0 (REPE) or 1 (REPNE)—you will want to determine which condition was responsible. The easiest way to find out what stopped the compare is to follow CMPS with a conditional transfer instruction that tests ZF: either JE (JZ) or JNE (JNZ).

For example, the following sequence makes the 8088 jump to NOT_FOUND if none of the first 100 elements of DEST and SOURCE match:

```
                CLD
                MOV     CX,100
REPNE           CMPS    DEST,SOURCE   ;Search for a match.
                JNE     NOT_FOUND     ;Matching elements found?
                ..                    ; Yes. Continue here.
                ..
NOT_FOUND:      ..                    ; No. Continue here.
                ..
```

Compare Byte Strings (CMPSB) and Compare Word Strings (CMPSW)

The assembler translates CMPS to either CMPSB (for bytes) or CMPSW (for words). You are encouraged to use these size-specific, no-operand forms instead of CMPS.

Scan-String Instructions

The scan-string instructions let you search a string in the extra segment for a specific value. The offset of the string's first element is in DI.

If you are scanning a byte string, the search value must be in AL. If you are scanning a word string, it must be in AX. The scan operation is nothing more than a compare-with-accumulator operation, and affects the flags in the same way as the string-compare instructions.

Scan String (SCAS)

The basic instruction in this group, *Scan String (SCAS)*, has the general form

```
SCAS  dest-string
```

where *dest-string* identifies a string in the extra segment whose offset is in DI. This operand tells the 8088 whether the search value is a byte in AL or a word in AX.

As with CMPS, to operate on more than one string element, apply the repeat prefix REPE (REPZ) or REPNE (REPNZ). For example, this sequence

```
        CLD
        LEA    DI,ES:B_STRING
        MOV    AL,' '
        MOV    CX,100
REPE    SCAS   B_STRING
```

searches up to 100 elements of the byte string *B_STRING*, looking for an element other than a space. If it finds such an element, the offset of the *next* element is returned in DI and ZF is 0. A subsequent JCXZ instruction tells whether the element was found (jump not taken) or not found (jump taken).

Scan Byte String (SCASB) and Scan Word String (SCASW)

The assembler translates SCAS to either SCASB (for bytes) or SCASW (for words). You are encouraged to use these size-specific, no-operand forms instead of SCAS.

Load-String and Store-String Instructions

Once you have located a string element with a compare-string or scan-string instruction, you normally want to perform some operation on it. You may want to read the element into a register (perhaps to determine its exact value) or change it in memory. The load-string and store-string instructions provide these operations.

Load String (LODS)

Load String (LODS) transfers a *source-string* operand addressed by SI from the data segment to AL (byte operation) or AX (word operation), then adjusts SI to point to the next element. SI is incremented if DF is 0, or decremented if DF is 1.

For example, the following sequence compares the 500-byte strings DEST and SOURCE to find the first non-matching elements. If a mismatch is detected, the SOURCE string element is loaded into AL.

```
        CLD
        LEA     DI,ES:DEST      ;Get offset of DEST
        LEA     SI,SOURCE       ; and SOURCE.
        MOV     CX,500          ;Element count.
REPE    CMPSB                   ;Search for a mismatch.
        JCXZ    MATCH           ;Mismatch found?
        DEC     SI              ; Yes. Adjust SI,
        LODS    SOURCE          ; read element into AL,
        ..                      ; and process it.
        ..
MATCH:  ..                      ; No mismatch. Continue
        ..                      ;  here.
```

Because this is a byte operation, the LODS instruction either increments SI by 1 (if DF = 0) or decrements SI by 1 (if DF = 1).

As usual, LODS has the optional shorter forms *Load Byte String (LODSB)* and *Load Word String (LODSW)*.

Store String (STOS)

Store String (STOS) transfers the byte in AL or word in AX to the *destination-string* operand in the extra segment, addressed by DI, then adjusts DI to point to the next element. DI is incremented if DF is 0, and decremented if DF is 1.

As a repeated operation, STOS is convenient for filling a string with a given value. For instance, the following sequence scans the 200-word string *W_STRING* for the first non-zero element. If such an element is found, this word and the next five words are filled with zeros.

```
        CLD
        LEA    DI,ES:W_STRING   ;Address string.
        MOV    AX,0             ;Search value is 0.
        MOV    CX,200           ;Search count is 200 words.
REPNE   SCASW                   ;Search the string.
        JCXZ   ALL0             ;Non-zero word found?
        SUB    DI,2             ; Yes. Adjust DI,
        MOV    CX,6             ;  then fill six words
REP     STOS   W_STRING         ;  with 0.
ALL0:   ..                      ; No. Continue here.
        ..
```

3.9 Interrupt Instructions

Like a procedure call, an interrupt makes the 8088 save return information on the stack, then jump to an instruction sequence elsewhere in memory. However, a procedure call makes the 8088 execute a procedure, while an interrupt makes it execute an *interrupt service routine*.

Although procedure calls can be NEAR or FAR and direct or indirect, an interrupt always makes an indirect jump to its service routine. It does this by obtaining the address of the routine from an *interrupt vector*, a 32-bit location in memory. Moreover, procedure calls save only an address on the stack, while interrupts also save the flags (as a PUSHF instruction does).

Interrupts can be activated by external devices in the system or by special interrupt instructions within a program. The 8088 has three different interrupt instructions—two "calls" and one "return"—as summarized in Table 3-14.

Table 3-14. Interrupt instructions.

Mnemonic	*Assembler Format*		*Flags*								
			OF	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
INT	INT	Interrupt-type	-	-	0	0	-	-	-	-	-
INTO	INTO		-	-	0	0	-	-	-	-	-
IRET	IRET		*	*	*	*	*	*	*	*	*

Note: - means unchanged and * means changed.

Interrupt (INT)

The INT instruction has the general form

```
INT  interrupt-type
```

where *interrupt-type* is the identification number of one of 256 different vectors in memory. (One vector for each of the 256 interrupts that we discussed in Chapter 1.)

When the 8088 executes INT, it does the following:

1. Push the Flags register onto the stack.
2. Clear the Trap Flag (TF) and the Interrupt Enable/Disable Flag (IF), to disable single-stepping and to "lock out" other maskable interrupts.
3. Push the CS register onto the stack.
4. Calculate the address of the interrupt vector, by multiplying *interrupt-type* by 4.
5. Load the second word of the interrupt vector into CS.
6. Push the IP onto the stack.
7. Load the first word of the interrupt vector into IP.

In summary, after INT has executed, the flags, CS, and IP are on the stack, TF and IF are 0, and the CS:IP combination points to the starting address of the interrupt service routine. Now the 8088 begins executing that service routine.

As we mentioned in Chapter 1, the 256 interrupt vectors are located in the lowest locations in memory. Since each is four bytes long, they occupy the first 1K bytes—absolute addresses 0 through 3FFH. For example, the instruction

```
INT  1AH
```

makes the 8088 calculate the vector address 68H (4 x 1AH). Thus, it obtains the 16-bit IP and CS values of the interrupt service routine from locations 68H and 6AH, respectively.

Figure 3-11 shows the stack, the Stack Pointer (SP), the Code Segment (CS) register, and the Instruction Pointer (IP) before and after this instruction is executed. In this example, we assume the interrupt vector holds the address F000:FE6E , so that is where the 8088 begins executing.

Of the 256 interrupt types, Intel has reserved the first five (Types 0 through 4) for internal interrupts. In the PC, many of the others are reserved for the computer's *Basic I/O System (BIOS)*, and for DOS and BASIC. We will discuss system interrupts in more detail in Chapter 6.

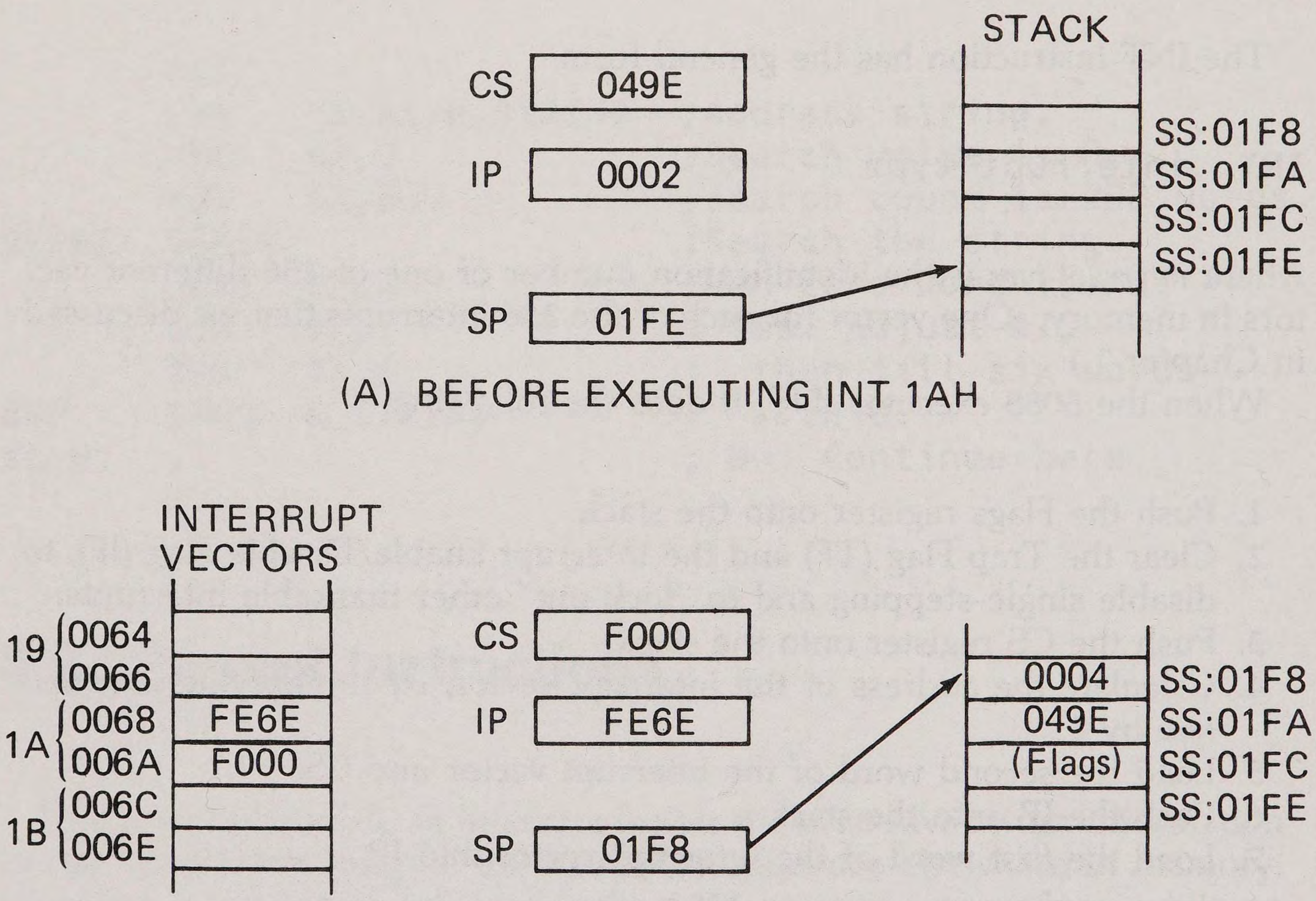


Figure 3-11. How an interrupt affects the stack.

Interrupt If Overflow (INTO)

Interrupt If Overflow (INTO) is a *conditional* interrupt instruction. INTO activates an interrupt only if the Overflow Flag (OF) is 1. In that case, INTO transfers control to an interrupt service routine with an indirect call through interrupt vector 4. (In other words, INTO activates a Type 4 interrupt.)

Interrupt Return (IRET)

Interrupt Return (IRET) is to interrupts what RET is to procedure calls. It "undoes" the work of the original operation and lets the 8088 make an orderly return to the main program. For this reason, IRET must be the last instruction the 8088 executes in an interrupt service routine.

IRET pops three 16-bit values off the stack and loads them into the Instruction Pointer (IP), Code Segment (CS) register, and the Flags register, respectively. Other program registers may be destroyed unless the interrupt service routine explicitly saves them.

3.10 Processor Control Instructions

These instructions let you regulate the operation of the 8088 from within a program. As Table 3-15 shows, there are three kinds of processor control instructions: flag operations, external synchronization, and the "do-nothing" instruction, NOP (no operation).

Table 3-15. Processor control instructions.

Mnemonic	*Assembler Format*	*Flags*								
		OF	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
Flag Operations										
STC	STC	-	-	-	-	-	-	-	-	1
CLC	CLC	-	-	-	-	-	-	-	-	0
CMC	CMC	-	-	-	-	-	-	-	-	*
STD	STD	-	1	-	-	-	-	-	-	-
CLD	CLD	-	0	-	-	-	-	-	-	-
STI	STI	-	-	1	-	-	-	-	-	-
CLI	CLI	-	-	0	-	-	-	-	-	-
External Synchronization										
HLT	HLT	-	-	-	-	-	-	-	-	-
WAIT	WAIT	-	-	-	-	-	-	-	-	-
ESC	ESC ext-opcode,source	-	-	-	-	-	-	-	-	-
LOCK	LOCK	-	-	-	-	-	-	-	-	-
No Operation										
NOP	NOP	-	-	-	-	-	-	-	-	-

Note: - means unchanged and * means changed.

Flag Operations

The 8088 has seven instructions that let you change the Carry Flag (CF), Direction Flag (DF), and Interrupt Enable Flag (IF).

Set Carry Flag (STC) and *Clear Carry Flag (CLC)* force CF to a 1 or 0 state, respectively. These are useful to prepare CF for an RCL or RCR rotate-with-carry operation. *Complement Carry Flag (CMC)* makes CF a 0 if it is a 1, and vice versa.

Set Direction Flag (STD) and *Clear Direction Flag (CLD)* force DF to a 1 or 0 state, respectively. You use STD and CLD to select the direction for a string operation. If DF is 0, the index registers SI and DI increment after each operation; if DI is 1, they decrement. For example,

```
      MOV    CX,100
      CLD
REP   MOVS   DEST,SOURCE
```

moves elements SOURCE through SOURCE+99 to the locations DEST through DEST+99. Conversely,

```
      MOV    CX,100
      STD
REP   MOVS   DEST,SOURCE
```

moves elements SOURCE through SOURCE-99 to locations DEST through DEST-99.

Clear Interrupt Flag (CLI) zeros IF, which makes the 8088 ignore maskable interrupts from external devices in the system. You generally disable such interrupts when the processor is performing some time-critical or high-priority task that cannot be interrupted. However, the processor will still process *non-maskable* interrupts while IF is 0.

Set Interrupt Flag (STI) sets IF to 1, which lets the 8088 respond to maskable interrupts from external devices.

External Synchronization Instructions

These instructions are used primarily to synchronize the 8088 with external events.

Halt (HLT) puts the 8088 into a halt state, in which it sits idle and executes no instructions. The 8088 leaves the halt state only if you reset it or it receives an external interrupt, either non-maskable or (if IF is 1) maskable. You can use HLT to make the processor wait for an interrupt (e.g., a character from the keyboard) before proceeding.

Wait (WAIT) also puts the 8088 into an idle state, but instead of just sitting there, it checks an input line called TEST at five-clock intervals. It will service interrupts during this time, but goes idle again upon return from the interrupt service routine.

If TEST is active, the 8088 proceeds to the instruction that follows WAIT. The purpose of WAIT is to stop the processor until some external device has completed its activity.

After reading how HLT and WAIT stop the processor, you might expect *Escape (ESC)* to send it on vacation! Not true. ESC simply makes it fetch the contents of a specified operand and put that operand on its data bus. Thus, ESC provides a way for other processors in the system to receive their instructions from the 8088 instruction stream.

The general form for ESC is:

```
ESC  ext-opcode,source
```

where *ext-opcode* is a 6-bit immediate number and *source* is a register or a memory variable. ESC is often used to send instructions to an 8087 Math Coprocessor (more about this in Chapter 12). Here, the 8087's op-code is *ext-opcode* and its operand is the contents of *source*.

Lock the Bus (LOCK) is a one-byte prefix that may precede any instruction. LOCK makes the 8088 activate its bus LOCK signal for as long as the LOCKed instruction takes to execute. While the LOCK signal is active, no other processor in the system can use the bus.

No Operation Instruction

The final instruction, *No Operation (NOP)*, is the simplest of all, because it does exactly what its name implies: it performs no operation whatsoever. NOP affects no flags, registers, or memory locations; it only advances the Instruction Pointer (IP).

Surprisingly, NOP has a variety of uses. For example, you can use its op-code (90H) to "patch" object code when you want to delete an instruction without reassembling the program. NOP is also convenient for testing sequences of instructions. That is, you can make NOP the last instruction in a test program—a convenient spot at which to stop a trace. You may find other uses for this innocuous, but handy, instruction.

3.11 Key Point Summary

Following are the key points you learned in this chapter:

1. The 8088 can use any of seven techniques or *modes* to obtain the data that an instruction is to operate on. The assembler tells it which mode to use based on the format of the operand in your source program.
2. The simplest addressing modes are *register* and *immediate,* because the 8088 obtains the operand directly from a register or from within the instruction itself. With the other five modes—*direct, register indirect, base relative, direct indexed,* and *base indexed*—the 8088 must calculate a memory address, then read the operand from the addressed location. Calculating the address may involve simply reading it from within an instruction, as in direct addressing. Then again, it may involve adding as many as three terms, as in base indexed addressing.
3. The 8088's *data transfer instructions* move data and addresses between registers or between a register and a memory location or I/O port. Instructions in this group can be further divided into four subgroups: *general-purpose, input/output, address transfer,* and *flag transfer.*
4. The most common general-purpose instruction, Move (MOV), copies a byte or word between a register and a memory location, or between two registers. It can also copy an immediate value (constant) into a register or memory location.

   Other useful general-purpose instructions are Push Word onto Stack (PUSH) and Pop Word off Stack (POP), which are useful for preserving the contents of a register while a program is using it for some other purpose.
5. The address transfer instructions include Load Effective Address (LEA), which loads the offset of a memory location into a register. LEA is common in string operations, where you are required to supply the offset of each string you want to use.
6. The 8088 has instructions that can perform arithmetic operations on signed or unsigned binary numbers and on packed or unpacked decimal numbers.
7. A packed decimal number has two digits per byte, while an unpacked decimal number has only one digit per byte. Each digit consists of four bits that represent a value between 0 and 9. Since the digits are decimal values rather than binary, we call them binary-coded decimal or *BCD.*
8. There are two separate add instructions. Add (ADD) can add single-byte or single-word numbers and the low-order terms of multi-precision numbers, while Add with Carry (ADC) can add only the higher-order terms of two multi-precision numbers. Increment Destination

by One (INC) adds one to an operand, but does not affect the Carry Flag (CF).

9. There are also two separate subtract instructions, Subtract (SUB) and Subtract with Borrow (SBB), as well as Decrement Destination by One (DEC).
10. Multiply instructions are Multiply (MUL) for unsigned numbers and Integer Multiply (IMUL) for signed numbers. Both take one operand from the instruction and obtain the second operand from either AL (for byte operations) or AX (for word operations). They return the product in either AH and AL or DX and AX.
11. Divide instructions are Divide (DIV) for unsigned numbers and Integer Divide (IDIV) for signed numbers. Both take the divisor from the instruction operand. If the divisor is a byte, they obtain the dividend from AH and AL, and return the quotient in AL and the remainder in AH. If the divisor is a word, they obtain the dividend from DX and AX, and return the quotient in AX and the remainder in DX.
12. The 8088 always performs arithmetic operations as if the operands were binary numbers. If you are adding, subtracting, or multiplying decimal (BCD) numbers, you must *adjust* the result. To do this, follow the ADD instruction with an ASCII Adjust for Addition (AAA) if the numbers are unpacked or with Decimal Adjust for Addition (DAA) if they are packed. Similarly, follow SUB with ASCII Adjust for Subtraction (AAS) or Decimal Adjust for Subtraction (DAS), or follow MUL with ASCII Adjust for Multiplication (AAM).

    To divide decimal numbers, you must first convert the unpacked dividend to binary. To do this, precede the DIV instruction with ASCII Adjust for Division (AAD).
13. The sign-extension instructions let you operate on mixed-size data. Convert Byte to Word (CBW) extends a byte in AL to a word in AX, while Convert Word to Doubleword (CWD) extends a word in AX to a doubleword in DX and AX.
14. The 8088's *bit manipulation group* includes logical, shift, and rotate instructions.
15. Logical instructions can take the Logical AND (AND), Inclusive-OR (OR), or Exclusive-OR (XOR) of two operands. A variation of AND, called Test (TEST), affects only the flags, not the operands. Finally, Logical NOT (NOT) takes the one's-complement of an operand.
16. Shift instructions displace an operand to the left or right. Use Shift Arithmetic Left or Right (SAL or SAR) to shift signed numbers; use Shift Logical Left or Right (SHL or SHR) to shift unsigned numbers. These instructions also make for faster multiply and divide operations.
17. Rotate instructions do the same as shifts, except they preserve displaced bits rather than discard them. Rotate Left (ROL) and Rotate Right (ROR)

enter a displaced bit into the opposite end of the operand; Rotate Left through Carry (RCL) and Rotate Right through Carry (RCR) put a displaced bit into CF and insert the CF bit into the operand.

18. *Control transfer instructions* can make the 8088 transfer to a different part of the program, either unconditionally or conditionally, or repeatedly execute a block of instructions—that is, *loop.*
19. There are three unconditional transfer instructions. Call a Procedure (CALL) transfers the 8088 to a procedure and Return from Procedure (RET) returns it to the calling program. Jump (JMP) makes it continue at a different part of the program.
20. The conditional transfer instructions make the 8088 transfer only if a specified condition is satisfied. They are usually preceded by a Compare (CMP) instruction, which sets up the flags.
21. Iteration control instructions establish repetitive loop operations based on a count in CX. Variations called Loop If Equal (LOOPE) and Loop If Not Equal (LOOPNE) provide for an alternate way out of the loop based on ZF = 1 or ZF = 0, respectively.
22. *String instructions* let you operate on blocks of consecutive bytes or words. In them, the 8088 always assumes that the destination string is in the extra segment and is addressed by DI, while the source string is the data segment and is addressed by SI. Generally, you put a repeat prefix (REP, REPE, or REPNE) in front of the string instruction to operate on consecutive elements.
23. The string instructions include Move (MOVS), Compare (CMPS), Scan (SCAS), Load (LODS), and Store (STOS). Each is available in three forms, one in which you specify the operand string(s) and two in which you specify only the size of the operands (bytes or words).
24. There are three *interrupt instructions.* Interrupt (INT) activates one of 256 interrupts based on the number you specify. The 8088 obtains the address of the interrupt service routine from a 32-bit *interrupt vector* in low memory. Interrupt Return (IRET) returns the processor to the calling program, just as RET does for a procedure. Interrupt If Overflow (INTO) is a conditional form of INT; it activates the interrupt only if the Overflow Flag (OF) is 1.
25. *Processor control instructions* let you regulate the 8088 from within a program. They include instructions that control the Carry, Direction, and Interrupt Enable Flags (CF, DF, and IF), and synchronize the 8088 with another processor. Also included is No Operation (NOP), which is useful for "patching" assembled object programs.

Study Exercises (answers on page 362)

1. Write an instruction that stores the contents of the AX register into a word location called SAVE_AX in the extra segment.

2. What does this sequence do?

```
MOV   AX,0
MOV   BX,AX
MOV   BP,AX
MOV   [BX],AX
MOV   [BP],AX
```

3. Which of the following instructions or sequences are invalid? (Assume variables are defined in the data segment and instructions are defined in the code segment.)

```
(a) K       EQU    1024
            ..
            ..
            MOV    K,AX

(b) TEMP    DB     ?
            ..
            ..
            MOV    AL,TEMP

(c) TEMP    DB     ?
            ..
            ..
            MOV    TEMP,AX

(d) TEMP    DB     ?
    T3      DB     10
            ..
            ..
            MOV    TEMP,T3

(e)         MOV    [BX][BP],AX
```

4. List two instructions that clear the AX register to zero.
5. How do these two instructions differ?

```
MOV   BX,OFFSET TABLE+4
LEA   BX,TABLE+4
```

6. Write a loop that subtracts a three-word variable called V2 from another three-word variable called V1.

7. What does this instruction do?

```
MUL 10
```

8. If AX contains 1234H and BX contains 4321H, list the contents of AX after each of these instructions is executed:

```
(a) AND   AX,BX
(b) OR    AX,BX
(c) XOR   AX,BX
(d) NOT   AX
(e) TEST  AX,BX
```

9. Write a sequence to *normalize* AX. That is, shift AX left until the most-significant "1" is in bit 15. If AX is initially zero or bit 15 already contains a 1, exit immediately.
10. What does this sequence do?

```
START:  MOV   CX,3
        SUB   AX,10
        LOOP  START
```

4

High-Precision Mathematics

If you have done any assembly language programming on one of the conventional 8-bit microprocessors, you are probably impressed with the arithmetic potential of the 8088. For starters, the very fact that the 8088 has built-in multiply and divide instructions means the hours (or days) you would normally spend developing multiplication and division programs are available for more stimulating activities, such as playing racquetball.

In this chapter we use the multiply and divide instructions as a basis for developing some programs that tackle tougher math problems. We will begin with programs that multiply 32-bit numbers. Then we will discuss how to handle overflow situations in divide operations, and conclude with a program that calculates square roots.

4.1 Multiplication

In Chapter 3 we studied the 8088's two multiplication instructions, Multiply, Unsigned (MUL) and Integer Multiply, Signed (IMUL). These instructions multiply byte- or word-length operands to produce double-length (16- or 32-bit) products.

Is it difficult to multiply numbers larger than 16 bits? Not at all, as you shall see. Anyone who has written a multiplication program for an 8-bit microprocessor knows that just *having* a multiply instruction of any kind makes up for whatever inconvenience you go through to extend its capabilities.

Unsigned 32-Bit × 32-Bit Multiply

The MUL instruction can handle only 8- or 16-bit operands, but you can use it to multiply multi-precision unsigned numbers. For instance, you can use it to multiply two 32-bit numbers. To do this, you calculate a series of 32-bit *cross products,* then combine them to form the final 64-bit product. You learned this method in elementary school to multiply decimal numbers with pencil and paper.

As you probably recall (in these days of pocket calculators, it may be a little hazy), you write the multiplicand with the multiplier below it, and perform a series of multiplications—one for each digit in the multiplier. Each individual multiplication produces a partial product, which you enter directly below the multiplier digit. Thus, each partial product is offset one digit position to the left of the preceding partial product.

For example, 124 times 103 looks like this:

```
   124     Multiplicand
× 103      Multiplier
-----
   372     Partial Product #1
  000      Partial Product #2
 124       Partial Product #3
-----
 12772     Final Product
```

Offsetting the partial products accounts for the *decimal weights* of the multiplier digits. In this example, the 3 is a "ones" digit, the 0 is a "tens" digit, and the 1 is a "hundreds" digit. Therefore, you could write the example as:

$$103 \times 124 = (3 \times 124) + (0 \times 124) + (100 \times 124)$$

or

$$103 \times 124 = (3 \times 1 \times 124) + (0 \times 10 \times 124) + (1 \times 100 \times 124)$$

In this section we will develop a short procedure that multiplies two 32-bit unsigned numbers and yields a 64-bit unsigned product. If you had no multiply instruction you would have to perform 32 separate multiplications, one for each bit in the multiplier.

Fortunately, the 8088 has an instruction that multiplies 16-bit unsigned numbers directly. This instruction, MUL, lets us treat a 32-bit multiplier and a 32-bit multiplicand as two 2-digit numbers, where each digit is 16 bits long. Thus, we can generate the 64-bit product with just *four* multiplications.

Figure 4-1 shows a symbolic representation of the multiplier (digits A and B) and the multiplicand (digits C and D), and illustrates how the four partial

products are derived and how they must be aligned to calculate the 64-bit final product. The circled numbers identify the four 16-bit additions you must make to calculate the final product. (For instance, Addition 1 adds the high 16 bits of Product #1 to the low 16 bits of Product #2.)

Using Figure 4-1 as a guide, we can develop a procedure that multiplies two 32-bit numbers. Example 4-1 shows such a procedure called MULU32 that obtains the multiplier and multiplicand from register pairs CX:BX and DX:AX, respectively. It returns the 64-bit unsigned product in these same four registers: DX (high 16 bits), CX (mid-upper 16 bits), BX (mid-lower 16 bits) and AX (low 16 bits). Example 4-1 also shows some scratch locations that you must set up in the data segment.

The MULU32 procedure is fairly straightforward if you refer to Figure 4-1 as you look at the instructions and their accompanying comments. To begin, MULU32 saves the multiplicand in memory, then generates the four 32-bit partial products. Once the partial products have been saved in memory, the only remaining step is to add them. Note that the Carry Flag stays intact between additions because MOV does not affect it.

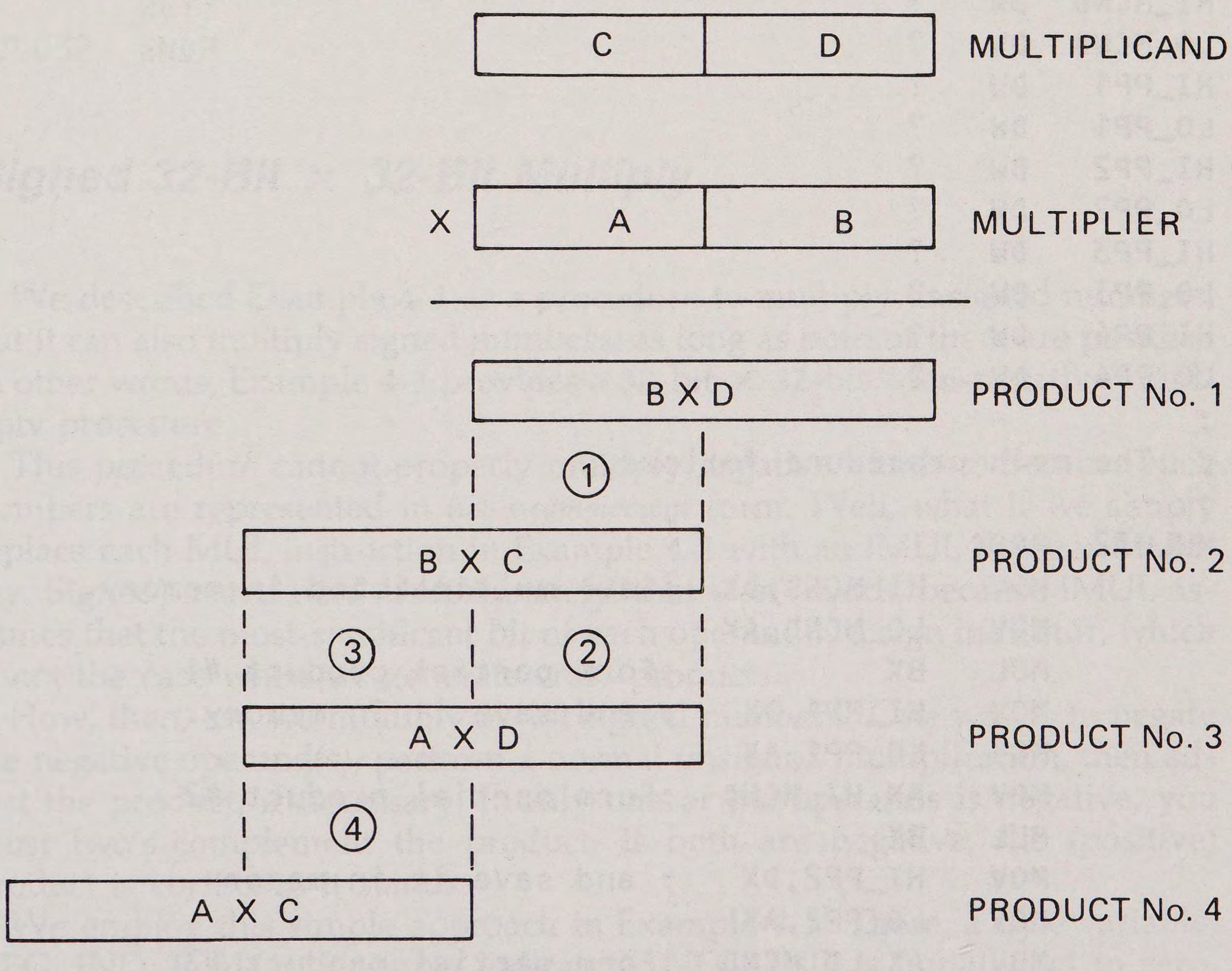


Figure 4-1. Generating a 64-bit product with four 16-bit by 16-bit multiplications.

Since a 32-bit operand can represent unsigned values as large as 4.294×10^9, the MULU32 procedure is satisfactory for most applications. (For those that involve larger numbers, you would probably use floating-point math!) However, it is certainly possible to develop a procedure that multiplies 64-bit (or longer) numbers using the cross-products approach.

Example 4-1. 32-bit × 32-bit unsigned multiply procedure.

```
;  This procedure multiplies two 32-bit unsigned numbers and
;  generates a 64-bit product.  Enter with the multiplier in CX
;  (high word) and BX (low word) and the multiplicand in DX
;  (high word) and AX (low word).  The product is returned in
;  DX, CX, BX and AX (high to low order).
;
;  Set up these variables in the data segment:
;
HI_MCND   DW   ?
LO_MCND   DW   ?
HI_PP1    DW   ?
LO_PP1    DW   ?
HI_PP2    DW   ?
LO_PP2    DW   ?
HI_PP3    DW   ?
LO_PP3    DW   ?
HI_PP4    DW   ?
LO_PP4    DW   ?
;
;  The main procedure follows.
;
MULU32  PROC
        MOV    HI_MCND,DX  ;Save multiplicand in memory.
        MOV    LO_MCND,AX
        MUL    BX          ;Form partial product #1
        MOV    HI_PP1,DX   ; and save it in memory.
        MOV    LO_PP1,AX
        MOV    AX,HI_MCND  ;Form partial product #2
        MUL    BX
        MOV    HI_PP2,DX   ; and save it in memory.
        MOV    LO_PP2,AX
        MOV    AX,LO_MCND  ;Form partial product #3
        MUL    CX
        MOV    HI_PP3,DX   ; and save it in memory.
        MOV    LO_PP3,AX
        MOV    AX,HI_MCND  ;Form partial product #4
```

```
        MUL     CX
        MOV     HI_PP4,DX       ; and save it in memory.
        MOV     LO_PP4,AX
;
;  Add the partial products to form the 64-bit final product.
;
        MOV     AX,LO_PP1       ;Low 16 bits.
        MOV     BX,HI_PP1       ;Form mid-lower 16 bits
        ADD     BX,LO_PP2       ; with sum #1.
        ADC     HI_PP2,0
        ADD     BX,LO_PP3       ; and sum #2.
        MOV     CX,HI_PP2       ;Form mid-upper 16 bits
        ADC     CX,HI_PP3       ; with sum #3
        ADC     HI_PP4,0
        ADD     CX,LO_PP4       ; and sum #4.
        MOV     DX,HI_PP4       ;Form high 16 bits
        ADC     DX,0            ; with propagated carry.
        RET
MULU32  ENDP
```

Signed 32-Bit × 32-Bit Multiply

We described Example 4-1 as a procedure to multiply unsigned numbers, but it can also multiply signed numbers, as long as both of them are positive. In other words, Example 4-1 provides a 32-bit × 32-bit "non-negative" multiply procedure.

This procedure cannot properly multiply negative numbers because such numbers are represented in *two's-complement* form. Well, what if we simply replace each MUL instruction in Example 4-1 with an IMUL (Integer Multiply, Signed) instruction? Unfortunately, that won't work, because IMUL assumes that the most-significant bit of each operand is a sign indicator, which is not the case when we generate cross products.

How, then, can we multiply 32-bit signed numbers? One way is to negate the negative operand(s), perform a normal unsigned multiplication, then adjust the product, if necessary. If only one of the operands is negative, you must two's-complement the product. If both are negative, the (positive) product is correct as it stands.

We employ this simple approach in Example 4-2. There, a byte variable, NEG_IND, holds a "negative indicator." NEG_IND is initially set to zero, and stays that way if both operands are positive. If one of the operands is negative, we take the one's-complement of NEG_IND, which makes it all ones. If both operands are negative, we one's-complement NEG_IND *twice*, which makes it all zeros again.

Example 4-2. 32-bit × 32-bit signed multiply procedure.

```
;  This procedure multiplies two 32-bit signed numbers and
;  generates a 64-bit product.  Enter with the multiplier in CX
;  (high word) and BX (low word) and the multiplicand in DX
;  (high word) and AX (low word).  The product is returned in
;  DX, CX, BX, and AX (high to low order).
;  This procedure calls MULU32 (Example 4-1).
;
;  Set up this variable in the data segment:
;
NEG_IND  DB    ?
;
;  The main procedure follows.
;
EXTRN    MULU32:FAR          ;MULU32 is an external procedure.
MULS32   PROC
         MOV   NEG_IND,0     ;Negative indicator = 0.
         CMP   DX,0          ;Multiplicand negative?
         JNS   CHKCX         ; No.  Check multiplier.
         NOT   AX            ; Yes.  2s-complement multiplicand.
         NOT   DX
         ADD   AX,1
         ADC   DX,0
         NOT   NEG_IND       ;   and 1s-complement indicator.
CHKCX:   CMP   CX,0          ;Multiplier negative?
         JNS   GOMUL         ; No.  Go multiply.
         NOT   BX            ; Yes.  2s-complement multiplier
         NOT   CX
         ADD   BX,1
         ADC   CX,0
         NOT   NEG_IND       ;   and 1s-complement indicator.
GOMUL:   CALL  MULU32        ;Perform unsigned multiplication.
         CMP   NEG_IND,0     ;Does product have correct sign?
         JZ    DONE          ; Yes.  Exit.
         NOT   AX            ; No.  2s-complement product.
         NOT   BX
         NOT   CX
         NOT   DX
         ADD   AX,1
         ADC   BX,0
         ADC   CX,0
         ADC   DX,0
DONE:    RET
MULS32   ENDP
```

Each time we one's-complement NEG_IND, one of the operands is negated (two's-complemented). Because the NEG instruction operates only on byte or word values, we must two's-complement our 32-bit operands in "brute-force" fashion. This involves one's-complementing the operand, then adding 1.

The MULS32 procedure calls MULU32 to perform the 32-bit by 32-bit multiplication. Since MULU32 is in another assembly module, we must declare it EXTRN at the start of the example. (Remember, the MULU32 module must have a PUBLIC MULU32 statement.) Upon return from MULU32, the state of NEG_IND determines whether the product is correct (NEG_IND zero) or needs negating (NEG_IND nonzero).

The execution time of the MULS32 procedure depends on whether the operands are both positive, both negative, or of the opposite sign. The following is a summary of these execution times in both clock cycles and microseconds:

Operands	Maximum Time (Cycles)	Maximum Time (μs)
Both positive	1096	230.16
Opposite signs	1136	238.56
Both negative	1144	240.24

4.2 Division

There are many applications for division, but one of the most common is taking the average of a set of numbers—perhaps the results of a series of laboratory tests. Example 4-3 shows a typical averaging program.

This procedure, AVERAGE, averages a specified number of unsigned word values pointed to by BX; the word count is contained in CX. It returns the integer portion of the average in AX and the fractional remainder in DX. For example, you could use the following sequence to average a 100-word table called TABLE:

```
LEA    BX,TABLE   ;Fetch offset of TABLE
MOV    CX,100     ; and its word count.
CALL   AVERAGE    ;Calculate the average.
```

In describing the DIV and IDIV instructions (Section 3.5), we mentioned that a divide operation automatically aborts if the divisor is zero or if *overflow* occurs. Overflow occurs when the dividend is so much larger than the divisor that the result register can't hold the quotient. An unsigned division overflows if the dividend is more than 65,535 times greater than the divisor.

Both divide-by-zero and overflow make the 8088 activate a Type 0 (Divide by Zero) interrupt. This interrupt saves the register contents, displays the message

Divide Overflow

then terminates the program.

The divide operation in Example 4-3 aborts if CX holds zero upon entry. Can an overflow also cause it to abort? No, overflow cannot occur here because the ratio of the dividend (word total) to the divisor (word count) can never exceed 65,536! However, some divide operations (e.g., dividing 200,000 by 2) may produce an overflow. Let's look at a procedure that always gives a valid quotient, irrespective of overflow.

Example 4-3. A word-averaging procedure.

```
;  This procedure takes the average of a specified number of
;  unsigned word values in the data segment. The offset of the
;  first word is contained in BX, the word count is contained
;  in CX.
;  It returns the integer portion of the average in AX and
;  the fractional remainder in DX.
;
AVERAGE  PROC
         SUB   AX,AX      ;Clear dividend to start.
         SUB   DX,DX
         PUSH  CX         ;Save word count on stack.
ADD_W:   ADD   AX,[BX]    ;Add next word to total
         ADC   DX,0
         ADD   BX,2       ; and update the index.
         LOOP  ADD_W      ;All words now totaled?
         POP   CX         ; Yes.  Retrieve word count
         DIV   CX         ;  and take the average.
         RET
AVERAGE  ENDP
```

Dealing with Overflow

In some applications, overflow signifies an error. In others, overflow is acceptable, but it means that the program must be able to accommodate a quotient longer than 16 bits. Since the division aborts when the 8088 encounters an overflow condition, how can you obtain a longer quotient? One easy way is to split the 32-bit dividend into two 16-bit numbers, then perform two 16-bit by 16-bit divide operations (which *cannot* produce an overflow).

If the divisor is a 16-bit number called X and the dividend is a 32-bit number represented by Y_1Y_0, you can represent the divide operation as

$$X \overline{)Y_1Y_0}$$

or, more properly, as

$$X \overline{)(Y_1 \times 2^{16}) + Y_0}$$

This division produces two 16-bit quotient digits (Q_1 and Q_0) and two 16-bit remainder digits (R_1 and R_0), as follows:

$$\begin{array}{r} Q_1 \times 2^{16} \\ X \overline{)Y_1 \times 2^{16}} \text{ and } R_1 \times 2^{16} \end{array}$$

$$\begin{array}{r} Q_0 \\ X \overline{)(R_1 \times 2^{16}) + Y_0} \text{ and } R_0 \end{array}$$

As you see, the combination of these two operations produce a 32-bit quotient, Q_1Q_0, and a 16-bit remainder, R_0. (The interim remainder R_1, if there is one, disappears during the second divide operation.) If no overflow occurs, Q_1 is zero, and the result is returned as $0Q_0$ and R_0.

We can use this approach to develop a divide procedure that *always* returns a valid quotient and remainder, regardless of overflow. Example 4-4 gives a procedure called DIVUO that does the job. It divides a 32-bit dividend in DX (high word) and AX (low word) by a 16-bit divisor in BX, and produces a 32-bit quotient in BX:AX and a 16-bit remainder in DX.

DIVUO is comprised of four steps:

1. Check whether the divisor in BX is zero. If so, call the Type 0 interrupt to abort the operation.
2. Change the Type 0 interrupt vector in absolute locations 0 (offset) and 2 (segment) so that it points to a new interrupt service routine—the one labeled OVR_INT in Example 4-4.
3. Perform the division. In the absence of overflow, the 8088 continues to the next consecutive instruction (SUB BX,BX). If overflow occurs, it activates the Type 0 interrupt service routine, which is now OVR_INT.
4. Restore the original Type 0 interrupt vector from values placed on the stack.

Example 4-4. Division procedure that accounts for overflow.

```
;  This divide procedure determines the correct quotient and
;  remainder, regardless of overflow.  Enter with the 16-bit
;  divisor in BX and the 32-bit dividend in DX (high word) and
```

```
;  AX (low word).
;  The 32-bit quotient is returned in BX:AX and the 16-bit
;  remainder is returned in DX.
;
DIVUO     PROC
          CMP   BX,0                    ;Divisor = 0?
          JNZ   DVROK
          INT   0                       ; Yes.  Abort the divide.
DVROK:    PUSH  ES                      ;Save current ES register,
          PUSH  DI                      ; DI register
          PUSH  CX                      ; and CX register.
          MOV   DI,0                    ;Fetch current INT 0 vector
          MOV   ES,DI
          PUSH  ES:[DI]                 ; and save it on the stack.
          PUSH  ES:[DI+2]
          LEA   CX,OVR_INT              ;Make INT 0 vector
          MOV   ES:[DI],CX              ; point to OVR_INT.
          MOV   CX,SEG OVR_INT
          MOV   ES:[DI+2],CX
          DIV   BX                      ;Perform the division.
          SUB   BX,BX                   ;If no overflow, make BX zero
RESTORE:  POP   ES:[DI+2]               ;Restore INT 0 vector.
          POP   ES:[DI]
          POP   CX                      ;Restore original CX,
          POP   DI                      ; DI
          POP   ES                      ; and ES.
          RET
;
;  This interrupt service routine is executed if the divide
;  operation produces overflow.
;
OVR_INT:  POP   CX                      ;Modify return address offset
          LEA   CX,RESTORE              ; to skip SUB BX,BX.
          PUSH  CX
          PUSH  AX                      ;Save current AX register.
          MOV   AX,DX                   ;Set up first dividend, 0-Y1.
          SUB   DX,DX
          DIV   BX                      ;Q1 is in AX, R1 is in DX.
          POP   CX                      ;Fetch original AX into CX.
          PUSH  AX                      ;Save Q1 on the stack.
          MOV   AX,CX                   ;Set up second dividend, R1-Y0
          DIV   BX                      ;Q0 is in AX, R0 is in DX.
          POP   BX                      ;Final quotient is in BX:AX.
          IRET
DIVUO     ENDP
```

With or without overflow, the DIVUO procedure returns a 32-bit quotient (BX:AX) and a 16-bit remainder (DX). If no overflow occurs, BX is zero.

4.3 Square Root

In this final section we will develop a program to calculate the integer square root of a 32-bit unsigned number, using the classical method of successive approximations known as Newton's method. This method says that *if A is an approximation for the square root of a number N, then*

$$A1 = (N/A + A)/2$$

is a better approximation.

To illustrate, suppose you want to find the square root of a number that has the value *N*. To get the first approximation, use the formula (N/200) + 2. To get the second approximation, divide N by the first approximation, then average the two results. To get the third approximation, divide N by the second approximation and average, and so on. For example, to find the square root of 10,000:

```
N = 10,000; first approximation is (10,000/200) + 2, or 52
   10,000/52 = 192, (192 + 52)/2 = 122
   10,000/122 = 81, (122 + 81)/2 = 101
   10,000/101 = 99, (101 + 99)/2 = 100
   10,000/100 = 100
```

So the square root of 10,000 is 100. We know that 100 *is* the square root, rather than just another intermediate approximation, because 100 times 100 is 10,000.

This particular number, 10,000, happens to have an integer square root, but not many numbers do. For example, the square root of 9999 is not an integer. Thus, if we try to computerize the successive approximation method and use

```
root × root = number
```

as the "all done" criteria, the processor will repeat the approximation instructions indefinitely, because the square of the *integer* approximation will never be 9999. Surely there must be a better way to stop the processor once it has found the closest, or "best," square root for a number.

You can use several different methods to end the approximation procedure. The one that best suits your needs depends on how accurate your answer must be and how much execution time you are willing to invest to get that answer.

For instance, you can let the 8088 execute the loop 10 times, and assume that that answer is accurate enough. Although it satisfies many applications, this

method is rather arbitrary. For a more precise solution, you can let the 8088 repeat the loop until it finds two successive approximations that are identical, or until they differ by only one. We take the latter approach in our example.

Example 4-5 gives a procedure (SQRT32) that uses successive approximations to calculate the integer square root of a 32-bit number. It obtains the source number from DX (high word) and AX (low word), and returns the 16-bit square root in BX.

To begin, the procedure saves BP, DX, and AX on the stack, then copies the Stack Pointer (SP) into BP; now BP points to AX on the stack. It then derives the first approximation using the formula (N/200) + 2.

The instruction at NXT_APP starts a loop that extends to the label DONE. With each pass through the loop, the 8088 calculates a new approximation by dividing the 32-bit source number (read from the stack) by the preceding approximation, then averaging the two results. It averages results by right-shifting AX, which effectively divides it by two. Using SHR instead of DIV here saves quite a bit of execution time—2 cycles for SHR versus a minimum of *80* cycles for DIV!

Example 4-5. Square root of a 32-bit number.

```
;  This procedure calculates the square root of a 32-bit integer
;  in DX (high word) and AX (low word), and returns that square
;  root as a 16-bit integer in BX.  The original number in DX:AX
;  is not affected.
;
SQRT32     PROC
           PUSH   BP             ;Save contents of BP
           PUSH   DX             ; and source number DX:AX.
           PUSH   AX
           MOV    BP,SP          ;BP points to AX on the stack.
           MOV    BX,200         ;As a first approximation,
           DIV    BX             ; divide source number by 200,
           ADD    AX,2           ; then add 2.
NXT_APP:   MOV    BX,AX          ;Save this approximation in BX.
           MOV    AX,[BP]        ;Read source number again.
           MOV    DX,[BP+2]
           DIV    BX             ;Divide it by last approximation.
           ADD    AX,BX          ;Average the last two results.
           SHR    AX,1
           CMP    AX,BX          ;Last two approximations identical?
           JE     DONE
           SUB    BX,AX          ; No.  Check for difference of 1.
           CMP    BX,1
           JE     DONE
```

```
                CMP     BX,-1
                JNE     NXT_APP
DONE:           MOV     BX,AX           ;Put result in BX.
                POP     AX              ;Restore source number
                POP     DX
                POP     BP              ; and scratch register BP.
                RET
SQRT32          ENDP
```

The procedure checks each new approximation against the preceding one, looking for approximations that are identical or differ by only 1 (+1 or -1). If either of these conditions occur, the processor transfers to DONE; otherwise it goes back to NXT_APP to calculate a new approximation. At DONE, the 8088 puts the final square root value in BX, then pops the source number (AX and DX) and the original value of BP off the stack.

Study Exercise (answer on page 363)

1. An interesting observation made a few years ago gives us a simple way to calculate square roots. The observation is this: *The square root of an integer is equal to the number of successively higher odd numbers that can be subtracted from it.*

   Figure 4-2 shows how you can extract the square root of 25 using this method. (Skeptics will want to try a few additional cases.) In this example, a total of five odd numbers—1, 3, 5, 7, and 9—can be subtracted from 25, so the square root is 5.

   Develop a procedure that employs this algorithm to take the square root of the 32-bit unsigned number in DX (high word) and AX (low word), and returns the 16-bit square root in BX. As with Example 4-5, AX and DX should be returned intact.

```
  25
- 1     Partial square root = 1
----
  24
- 3     Partial square root = 2
----
  21
- 5     Partial square root = 3
----
  16
- 7     Partial square root = 4
----
   9
- 9     Square root         = 5
----
   0
```

Figure 4-2. Obtaining a square root with odd-number subtractions.

5

Operating on Data Structures

There are almost as many ways to organize information in memory as there are kinds of information to be organized. These organizational techniques vary with the application, and have such names as lists, arrays, strings, and look-up tables. These are all different kinds of *data structures.*

The subject of data structures can (and does) fill many volumes, so we can't hope to give it an exhaustive treatment in this book. Instead, we will concentrate on three basic structures: *lists, look-up tables* and *text files.*

Lists hold units of data (bytes or words), called *elements,* arranged sequentially in memory. The sequence can be *consecutive,* where elements occupy adjacent memory locations, or *linked,* where each element contains a "pointer" to the next one in the list. Moreover, the elements may be arranged randomly or in ascending or descending order.

Look-up tables are data structures that hold information (either data or addresses) that has a defined relationship to a known value. A telephone directory is a look-up table; knowing a name, you can look up an associated telephone number.

Text files consist of non-numeric information such as letters, reports, and telephone lists.

5.1 Unordered Lists

In our ordered society, where telephone book listings are arranged alphabetically and house numbers increase or decrease as you go up or down a street, unordered *anythings* somehow seem bothersome. Still, not everything can be neatly ordered, so unordered lists remain a fact of life in many applications, particularly those that involve random data or data that change with time. For example, computerized weather stations may store hourly temper-

ature readings in unordered lists and manufacturers may log monthly shipping statistics in them.

Most lists are comprised of an element count byte (or word) and one or more data elements. When you work with a list, you generally want to add or delete elements, or search for one of a certain value. These operations are fairly easy to do.

1. To add an element, store it at the end of the list and add 1 to the element count.
2. To delete an element, move the rest of the elements upward in memory, then subtract 1 from the element count.
3. To search for an element value, compare each element to the search value, starting with the first element in the list.

Adding an Element to an Unordered List

Procedure ADD_TO_UL, shown in Example 5-1, is the kind of program used to create an unordered list or add an element to an existing one. In this case, the list contains word values (either signed or unsigned).

ADD_TO_UL reads the element count into CX, then scans the data elements for the value in AX. If this value is already in the list (the final value of ZF is 0), the 8088 pops the starting address back into DI and returns. Otherwise, it adds the value to the end of this list and increases the element count by one.

How long does this procedure take to execute? That depends on the number of elements and whether the search value is already in the list. The primary factor is how many times the Scan String (SCASW) instruction gets repeated. SCASW executes in (9 + 15N) cycles, where N is the number of repetitions. Let's examine the timing for both cases—value is not in the list and value is in the list—for a list that has N data elements.

If the search value is not in the list, the SCASW instruction executes N times. The remaining instructions in the procedure execute only once, and take 135 cycles. Therefore,

$$\begin{aligned}\text{Execution Time} &= 9 + 19N + 135\\ &= 19N + 144 \text{ cycles}\end{aligned}$$

Thus, adding an element to a 100-element list takes 2044 cycles, or 429.24 us.

Example 5-1. Adding an element to an unordered list.

```
;  This procedure adds the value in AX to an unordered list in
;  the extra segment, if it is not already in the list.
;  The starting address of the list is in DI.  The length of the
```

```
;  list, in words, is in the list's first location.
;  DI and AX are returned unaltered.
;
ADD_TO_UL   PROC
            CLD                     ;Make DF = 0, to scan forward.
            PUSH    DI              ;Save starting address.
            PUSH    CX
            MOV     CX,ES:[DI]      ;Fetch word count.
            ADD     DI,2            ;Make DI point to 1st data element.
REPNE       SCASW                   ;Value already in the list?
            POP     CX
            JNE     ADD_IT
            POP     DI              ; Yes.  Restore starting address
            RET                     ;  and exit.
ADD_IT:     STOSW                   ; No.  Add it to end of list,
            POP     DI              ;  then update element count.
            INC     WORD PTR ES:[DI]
            RET
ADD_TO_UL   ENDP
```

If the search value is in the list, it should take the 8088 an average of N/2 comparisons to locate it, because 50 percent of the time a search value will be in the lower half of the list and 50 percent of the time it will be in the upper half. This means scanning should take (9 + 9.5N) cycles, theoretically. The remaining instructions in the procedure take an additional 78 cycles. Therefore, on the average,

$$\begin{aligned}\text{Execution Time} &= 9 + 9.5N + 78 \\ &= 9.5N + 87 \text{ cycles}\end{aligned}$$

Thus, finding an element in an unordered 100-element list takes an average of 1037 cycles, or 217.77 μs.

Deleting an Element from an Unordered List

To delete an element from an unordered list, you must find it, then move the rest of the elements up one position. In doing this, you write over the deletion "victim." Since an element has been removed, you decrease the element count (the first location in the list) by one.

To illustrate, Figure 5-1(A) shows a list of bytes in memory. Since the list has six data elements, the first location (LIST) holds the value 6. Figure 5-1(B) shows what the list looks like after the fourth element (14) has been deleted. The list then has only five data elements and the values 97 and 8 have moved up in memory, eradicating the deleted value.

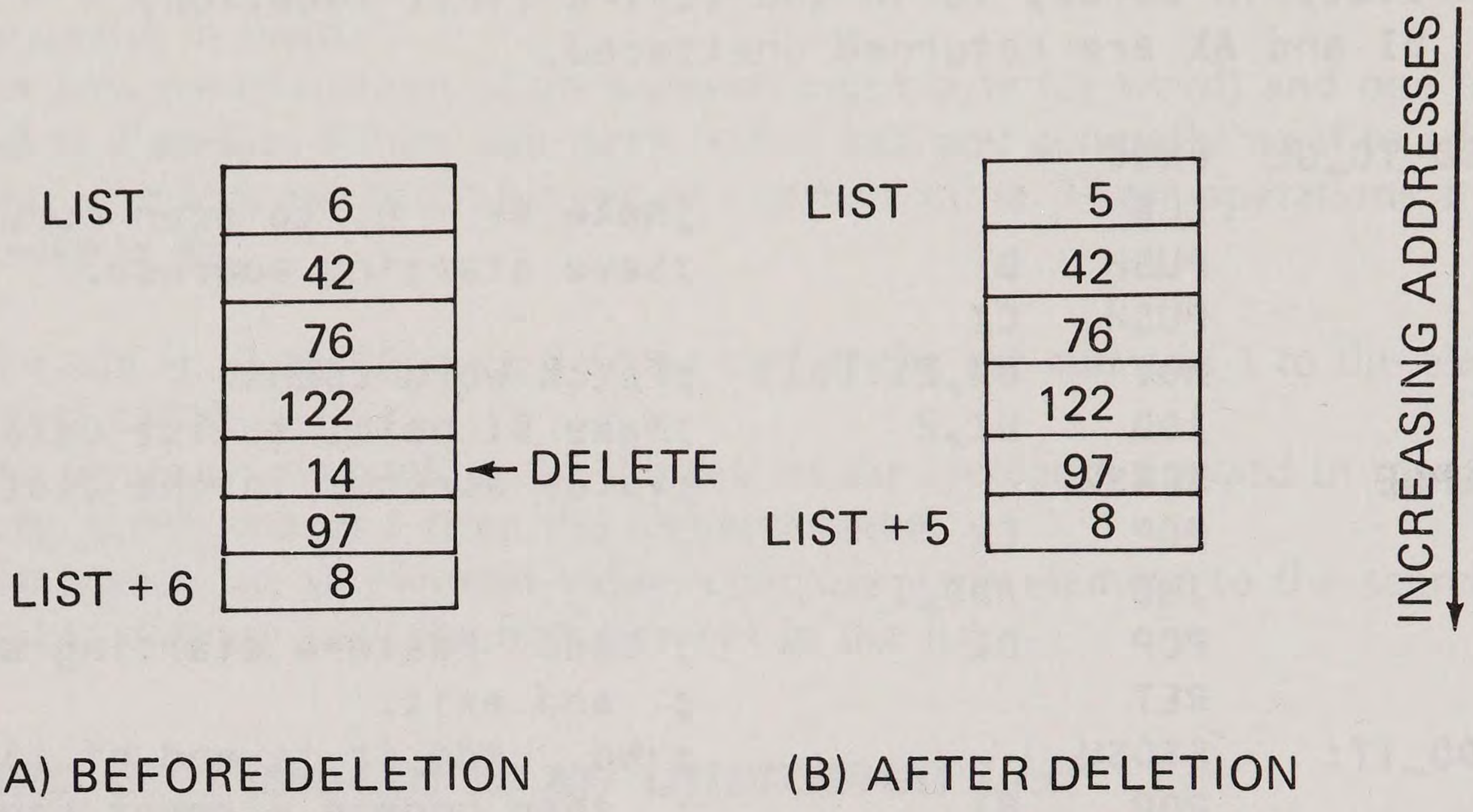


Figure 5-1. How a deletion affects a list.

The DEL_UL procedure in Example 5-2 performs just such an operation, using AX to specify the value to be deleted. As in Example 5-1, DI points to the start of the list.

The instructions that precede REPNE load the element count into CX and the address of the first data element into DI, then scan the list for the search value. These instructions are identical to the ones at the beginning of Example 5-1. If the search value is in the list (ZF = 1), the 8088 jumps to DELETE.

At DELETE, the processor takes one of two paths. If the element to be deleted is at the end of the list (CX contains zero), the 8088 jumps to DEC_CNT, where it simply decreases the list's element count. If the element to be deleted is anywhere else in the list, the loop at NEXT_EL moves all remaining elements up one position, overwriting the victim. The element count is then decreased by one to reflect the deletion.

Maximum and Minimum Values in an Unordered List

You may sometimes want to find the largest and smallest values in an unordered list. One way to find them is to initially declare the first element as both the maximum and the minimum value, then compare each of the remaining elements in the list to those values. If your program finds an element whose value is less than the minimum, it makes that value the new *minimum.* Similarly, if it finds an element whose value is greater than the maximum, it makes that value the new *maximum.*

Example 5-2. Deleting an element from an unordered list.

```
;  This procedure deletes the value in AX from an unordered
;  list in the extra segment, if it is in the list.
;  The starting address of the list is in DI.  The length of the
;  list, in words, is in the list's first location.
;  DI and AX are returned unaltered.
;
DEL_UL     PROC
           CLD                     ;Make DF = 0, to scan forward
           PUSH  BX                ;Save scratch register BX
           PUSH  DI                ; and starting address.
           MOV   CX,ES:[DI]        ;Fetch element count.
           ADD   DI,2
REPNE      SCASW                   ;Value in the list?
           JE    DELETE            ; If so, delete it.
           POP   DI                ; Otherwise, restore registers
           POP   BX
           RET                     ;  and exit.
;
;  The following instructions delete an element from the list,
;  as follows:
;     (1)  If the element is at the end of the list,
;          delete it by decreasing the element count by 1.
;     (2)  Otherwise, delete the element by moving all
;          subsequent elements up by one position.
;
DELETE:    JCXZ  DEC_CNT           ;If (CX) = 0, delete last element.
NEXT_EL:   MOV   BX,ES:[DI]        ;Move one element up in list.
           MOV   ES:[DI-2],BX
           ADD   DI,2              ;Point to next element.
           LOOP  NEXT_EL           ;Repeat until all are moved.
DEC_CNT:   POP   DI                ;Decrease element count by 1.
           DEC   WORD PTR ES:[DI]
           POP   BX                ;Restore contents of BX
           RET                     ; and exit.
DEL_UL     ENDP
```

The MINMAX procedure in Example 5-3 applies this method to an unordered list of unsigned words whose starting address is in DI. MINMAX returns the maximum and minimum values in AX and BX, respectively.

This procedure has three parts. The first part determines how many comparisons to make (element count - 1) and sets up the first data element as both the maximum and minimum norm. The second and third parts loop

through the list searching for a new minimum and a new maximum, respectively. New minimums are loaded into BX; new maximums are loaded into AX.

Although MINMAX processes lists of unsigned word values, you can easily modify it to search for the maximum and minimum in lists of signed words. In Example 5-3, simply replace JAE NOMIN with JGE NOMIN and JBE NOMAX with JLE NOMAX. For the reason behind this, see Table 3-11 in Section 3.7.

Example 5-3. Maximum and minimum values in an unordered list.

```
;  This procedure finds the maximum and minimum word values in ar
;  unordered list in the extra segment, and returns those values
;  in AX and BX, respectively.
;  The starting address of the list is in DI.  The length of the
;  list, in words, is in the list's first location.
;  DI is returned unaltered.
;
MINMAX   PROC
         PUSH  CX
         PUSH  DI                  ;Save starting address.
         MOV   CX,ES:[DI]          ;Fetch word count.
         DEC   CX                  ;Get ready for count-1 comparisons.
         PUSH  CX                  ;Save this count value.
         MOV   BX,ES:[DI+2]        ;To start, make 1st element minimum
         MOV   AX,BX               ; and maximum.
;
;  These instructions find the minimum value in the list.
;
         ADD   DI,4                ;Point to 2nd element in the list
         PUSH  DI                  ; and save this pointer.
CHKMIN:  CMP   ES:[DI],BX          ;Compare next element to minimum.
         JAE   NOMIN               ;New minimum found?
         MOV   BX,ES:[DI]          ; Yes.  Put it in BX.
NOMIN:   ADD   DI,2                ;Point to next element.
         LOOP  CHKMIN              ;Check entire list.
;
;  These instructions find the maximum value in the list.
;
         POP   DI                  ;Point to 2nd element in the list.
         POP   CX                  ;Reload comparison counter.
CHKMAX:  CMP   ES:[DI],AX          ;Compare next element to maximum.
         JBE   NOMAX               ;New maximum found?
```

```
        MOV   AX,ES:[DI]      ; Yes.  Put it in AX.
NOMAX:  ADD   DI,2            ;Point to next element.
        LOOP  CHKMAX          ;Check entire list.
        POP  DI
        POP  CX
        RET
MINMAX  ENDP
```

5.2 Sorting Unordered Data

If you are plotting information versus time or processing text, you can accept the information in unordered form. However, in many applications it is better to have the information arranged in increasing or decreasing order, because it is easier to analyze that way.

How can you rearrange a list of unordered data? There is a considerable amount of literature on this subject, but we'll concentrate on one common sorting technique called the *bubble sort.* If you want to investigate other sorting techniques, D. E. Knuth's classic *The Art of Computer Programming. Volume 3: Sorting and Searching* (Addison-Wesley) is an excellent starting point.

Bubble Sort

The bubble sort technique is so named because it makes list elements "rise" upward in memory (to higher-numbered addresses) like soap bubbles rise in the air. A bubble sort accesses elements in a list sequentially, starting with the first element, and compares each element to the next one in the list.

If the bubble sort program finds an element that is greater than its higher-addressed neighbor, it exchanges these elements. It then compares the next two elements, exchanges them if required, and so on. By the time the 8088 gets to the last element, the highest-valued element will have "bubbled-up" to that final list position.

In sorting with this algorithm, the processor usually makes several passes through the list, as you can see from the simple example in Figure 5-2. Here, the first pass "bubbles" 50 to the end of the list and the next two passes bubble 40 and 30 to the next highest positions. Therefore, this particular list gets sorted in three passes.

Seeing pass-by-pass "snapshots" of the list, as in Figure 5-2, makes it easy for *you* to know when a list is entirely sorted, but how can a *computer* know this? Unless you give it a specific pass count or tell it when to stop

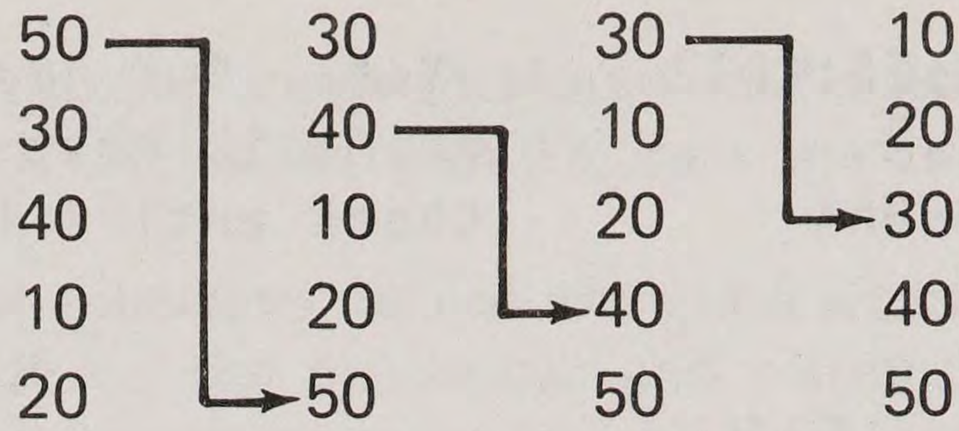


Figure 5-2. A bubble sort "bubbles" the largest numbers to the end.

in some other way, the computer will go merrily along, executing pass after pass, ad infinitum. Since the number of sorting passes depends on the initial arrangement of the list, we have no way of providing an exact pass count in a program. As an alternative, we will set up a special indicator called an *exchange flag* that the computer can use to find out when to stop sorting.

The exchange flag is set to 1 before each sorting pass. Any sorting pass that includes an element exchange operation changes it to 0. Therefore, after each pass, the value of the exchange flag tells the computer whether to continue sorting. A 0 tells it to make another pass through the list; a 1 indicates the list is sorted, and tells it to stop sorting. Figure 5-3 shows a flowchart of the bubble sort algorithm.

As you can see, even if a list is already in order at the outset, it takes the processor one pass to deduce this fact. If one pass is the minimum, what *maximum* number of passes may you anticipate? Since the five-number list in Figure 5-2 was already partially sorted, we made only three sorting passes to put it in ascending order. One more pass is needed to detect that this list is indeed sorted, making four passes altogether.

If that list had been initially arranged in descending order (the worst case), the processor would have made five passes through the list; four passes to sort the data and one more to determine no further sorting was needed. From this observation, we can state that *an N-element list takes from one to N passes to sort, with (N + 1)/2 passes being the average.*

A Bubble Sort Program

With the preceding background in bubble sort theory and a flowchart showing what needs to be done, we are prepared to write a program that sorts a list. Example 5-4 shows a procedure (B_SORT) that sorts a list of word values. (You can easily modify it to sort a list of bytes.)

As usual, the list is in the extra segment and its starting address is in DI. B_SORT uses BX to hold the exchange flag (which always contains either 1 or 0) and AX to hold the element value that is being compared to the next element in the list.

The B_SORT procedure also uses two word variables in the data segment: *SAVE_CNT* holds the comparison count (element count - 1) and

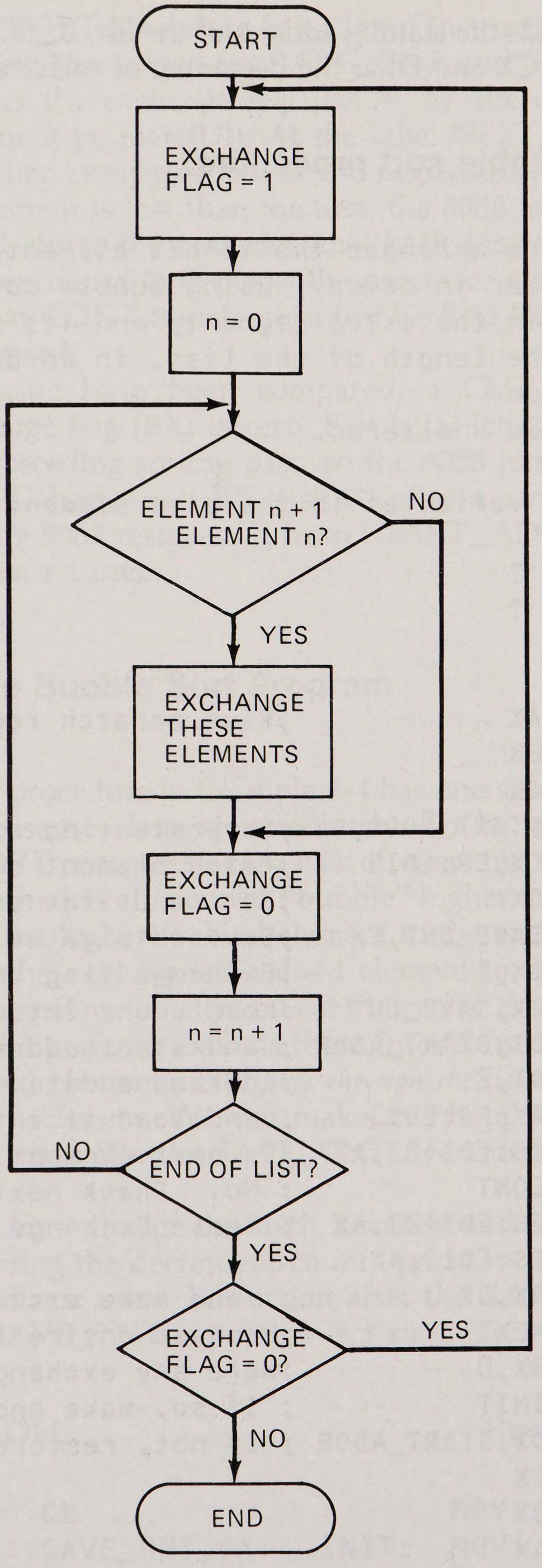


Figure 5-3. The bubble sort algorithm.

START_ADDR holds the starting address of the list. B_SORT uses these variables to reinitialize CX and DI at the beginning of each new sort operation.

Example 5-4. Bubble sort procedure.

```
;  This procedure arranges the 16-bit elements of a list in
;  ascending order in memory, using bubble sort.
;  The list is in the extra segment, and its starting address
;  is in DI.  The length of the list, in words, is in its
;  first location.
;  DI is returned unaltered.
;
;  Set up these variables in the data segment.
;
SAVE_CNT    DW  ?
START_ADDR  DW  ?
;
B_SORT   PROC
         PUSH  AX                ;Save scratch registers
         PUSH  BX
         PUSH  CX
         MOV   START_ADDR,DI ;Save starting address
         MOV   CX,ES:[DI]    ;Fetch element count
         DEC   CX            ;Get ready for count-1 compares
         MOV   SAVE_CNT,CX   ;Save this value in memory
INIT:    MOV   BX,1          ;Exchange flag (BX) = 1
         MOV   CX,SAVE_CNT   ;Load count into CX
         MOV   DI,START_ADDR ; and start address into DI
NEXT:    ADD   DI,2          ;Address a data element
         MOV   AX,ES:[DI]    ; and load it into AX
         CMP   ES:[DI+2],AX  ;Is next element < this one?
         JAE   CONT          ; No.  Check next pair
         XCHG  ES:[DI+2],AX  ; Yes.  Exchange these elements
         MOV   ES:[DI],AX
         SUB   BX,BX         ;  and make exchange flag 0
CONT:    LOOP  NEXT          ;Process entire list
         CMP   BX,0          ;Were any exchanges made?
         JE    INIT          ; If so, make another pass
         MOV   DI,START_ADDR ; If not, restore registers
         POP   CX
         POP   BX
         POP   AX
         RET                 ;  and exit.
B_SORT   ENDP
```

Although the B_SORT procedure has a lot of instructions, it is quite simple. After calculating the values for SAVE_CNT and START_ADDR, the procedure initializes the exchange flag (BX = 1), the comparison counter (CX), and the element pointer (DI). At the label NEXT, the 8088 loads an element into AX, then compares it with the next element in memory.

If the second element is less than the first, the 8088 loads the second element into AX and stores the two elements back into memory, in reverse order. Because an exchange took place, the processor clears BX to zero. The LOOP instruction at CONT transfers control back to NEXT until the entire list has been processed.

When all elements have been compared, a CMP instruction checks whether the exchange flag (BX) is zero. If it is, at least one exchange took place during the preceding sorting pass, so the 8088 jumps back to INIT to begin a new pass. Otherwise, if BX is still 1 after a sorting pass, the list is finally sorted, so the 8088 restores DI from START_ADDR and BX and AX from the stack, then returns.

Streamlining the Bubble Sort Program

The bubble sort procedure in Example 5-4 has one subtle, but noteworthy, deficiency: it sorts some elements needlessly. To be specific, during each sorting pass B_SORT compares every pair of elements in the list. However, note that each pass makes an element "bubble" higher in the list. That is, the first pass bubbles the highest-valued element to the end of the list, the second pass bubbles the next highest-valued element to the next-to-last position, and so on. Therefore, the elements at the end of the list are in their final (sorted) positions, so you may exclude them from subsequent sorting passes!

To exclude sorted elements, *each pass through the list should involve one less comparison than its predecessor.* We can make this happen by modifying our procedure so that the value in SAVE_CNT decrements before each new pass.

To do this, you must change the sixth, seventh, and eighth instructions in the procedure to bring the decrement-count operation under the label INIT. You must also add one more instruction after the decrement, to make the processor exit if SAVE_CNT is 0. Here is a summary of the changes:

Old

```
        DEX   CX
        MOV   SAVE_CNT,CX
INIT:   MOV   BX,1
```

Revised

```
        MOV   SAVE_CNT,CX
INIT:   MOV   BX,1
        DEC   SAVE_CNT
        JZ    SORTED
```

Here, SORTED refers to the MOV instruction that restores the contents of DI from START_ADDR. Example 5-5 shows a new bubble sort procedure (BUBBLE) that has these changes.

For any given list, BUBBLE makes the same number of sorting passes as the B_SORT procedure in Example 5-4. But because BUBBLE makes about half as many *comparisons,* it executes much faster than B_SORT.

For example, to sort 100 elements arranged in decreasing order, B_SORT makes 100 passes, with 99 comparisons in each pass—a total of 9,900 comparisons. By contrast, BUBBLE makes 100 passes, with 99 comparisons in the first pass and one in the last pass, for an average of 50 comparisons—a total of 5,500 comparisons.

To compare BUBBLE and B_SORT, I sorted two lists of 16-bit elements using both procedures. Both lists were initially arranged in decreasing order. The first list, which had 500 elements, was sorted in 7.5 seconds by B_SORT and in 4.5 seconds by BUBBLE. The second list, which had 1000 elements, was sorted in 28.0 seconds by B_SORT and in 16.5 seconds by BUBBLE. Based on these tests, we can conclude that *BUBBLE sorts a list about 40 percent faster than B_SORT.*

Note that the sorting time rises dramatically as you take on longer lists. The bubble sort procedures presented here can sort lists up to 32K words long, but you'd better be prepared to wait if your list is that extensive. In fact, a "worst-case" list of even 2,000 words takes BUBBLE about 66 seconds to sort.

Example 5-5. A better bubble sort procedure.

```
;  This procedure arranges the 16-bit elements of a list in
;  ascending order in memory, using bubble sort.
;  The list is in the extra segment, and its starting address
;  is in DI.  The length of the list, in words, is in its
;  first location.
;  DI is returned unaltered.
;
;  Set up these variables in the data segment.
;
SAVE_CNT     DW  ?
START_ADDR   DW  ?
;
BUBBLE    PROC
          PUSH  AX                  ;Save scratch registers
          PUSH  BX
          PUSH  CX
          MOV   START_ADDR,DI ;Save starting address
          MOV   CX,ES:[DI]    ;Fetch element count
          MOV   SAVE_CNT,CX   ;Save this value in memory
```

```
INIT:     MOV   BX,1              ;Exchange flag (BX) = 1
          DEC   SAVE_CNT          ;Get ready for count-1 compares
          JZ    SORTED            ;Exit if SAVE_CNT is 0
          MOV   CX,SAVE_CNT       ;Load the compare count into CX
          MOV   DI,START_ADDR     ; and the start address into DI
NEXT:     ADD   DI,2              ;Address a data element
          MOV   AX,ES:[DI]        ; and load it into AX
          CMP   ES:[DI+2],AX      ;Is next element < this one?
          JAE   CONT              ; No.  Check next pair
          XCHG  ES:[DI+2],AX      ; Yes.  Exchange these elements
          MOV   ES:[DI],AX
          SUB   BX,BX             ;  and make exchange flag 0
CONT:     LOOP  NEXT              ;Process entire list
          CMP   BX,0              ;Were any exchanges made?
          JE    INIT              ; If so, make another pass
SORTED:   MOV   DI,START_ADDR     ; If not, restore registers
          POP   CX
          POP   BX
          POP   AX
          RET                     ;  and exit.
BUBBLE    ENDP
```

5.3 Ordered Lists

Now that you know how to sort a list, let us discuss how to search for a known value, and then see how to add and delete elements.

Searching an Ordered List

In Section 5.1 we learned that locating a value in an unordered list requires searching through it element by element. This takes an average of N/2 comparisons for an N-element list. If a list is *ordered*, however, you can use any of several search techniques to find a value. For all but the shortest lists, most of these techniques are faster and more efficient than searching sequentially.

Binary Search

One common technique for searching ordered lists is the *binary search*. Its name reflects the fact that it divides the list into a series of progressively shorter halves ("bi" is Latin for "two"), and eventually closes in on one element. A binary search starts in the middle of the list and determines whether

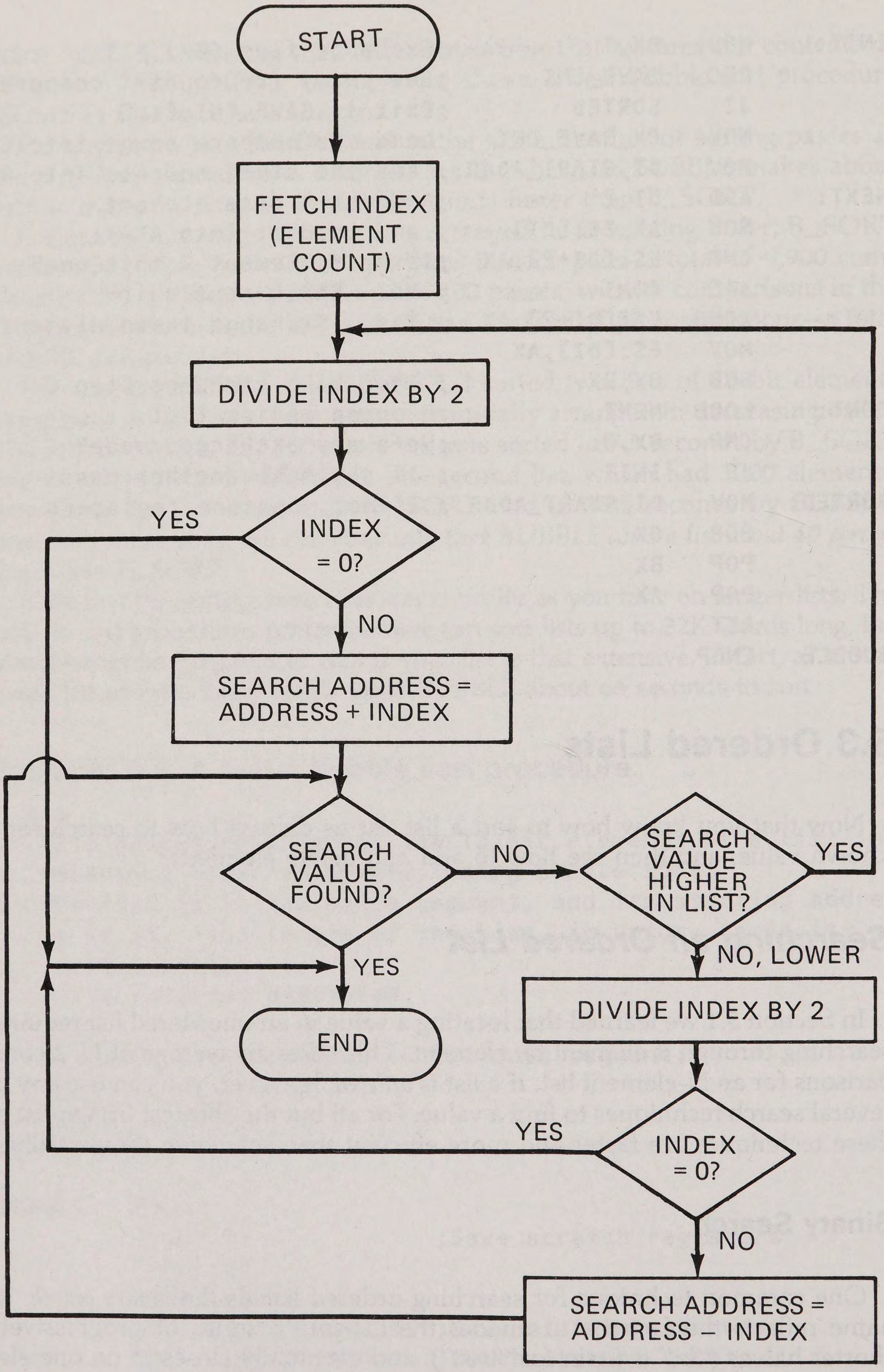


Figure 5-4. Binary search algorithm.

the search value lies above or below that point. It then takes *that* half of the list and divides *it* into halves, and so on.

The flowchart in Figure 5-4 shows how to make a binary search on an ordered list, producing an address as the result. If the search value is in the list, the address is that of the matching element. If the value is not in the list, the address is that of the last location to be compared. Of course, the program that performs this search must also return some kind of indicator that tells you whether the address reflects a successful or unsuccessful search.

Example 5-6 shows a procedure (B_SEARCH) you can use to search an ordered list of unsigned words. The fact that this procedure operates on words instead of bytes requires us to make a few changes to our basic algorithm. For one thing, because words lie two bytes apart in memory, we must include instructions that ensure the index is always a multiple of two; that is, an *even* value. For the same reason, we terminate a search, and declare it unsuccessful, when the index has decreased to 2 (instead of 0).

In this procedure, the search value is in AX and the starting address of the list is in DI. B_SEARCH returns the result address in SI, and CF tells whether the value was found (CF = 0) or not found (CF = 1).

Example 5-6. 16-bit binary search procedure.

```
;  This procedure searches an ordered list in the extra segment
;  for the word value contained in AX.
;  The starting address of the list is in DI.  The length of the
;  list, in words, is in the list's first location.
;  The results are returned in SI and the Carry Flag (CF), as
;  follows:
;       1.  If the value is in the list, CF is 0 and SI
;           holds the address of the matching element.
;       2.  If the value is not in the list, CF is 1 and
;           SI holds the address of the last element to
;           be compared.
;  In either case, AX and DI are unaffected.
;
;  Set up this variable in memory.
;
START_ADDR  DW  ?
;
B_SEARCH    PROC
;
;  First find out if AX lies beyond the boundaries of the list.
;
            CMP    AX,ES:[DI+2]     ;Search value < or = first el.?
            JA     CHK_LAST         ; No.  Check last element
```

```
                LEA     SI,ES:[DI+2]    ; Yes.  Fetch address of 1st el.
                JE      EXIT_1ST        ;If value = 1st element, exit
                STC                     ;If value < 1st element, set CF
EXIT_1ST:       RET
CHK_LAST:       MOV     SI,ES:[DI]      ;Point to last element
                SHL     SI,1
                ADD     SI,DI
                CMP     AX,ES:[SI]      ;Search value > or = last el.?
                JB      SEARCH          ; No.  Search list
                JE      EXIT_LAST       ; Yes.  Exit if value = element
                STC                     ;If value > last element, set CF
EXIT_LAST:      RET
;
;  Search for the value within the list.
;
SEARCH:         MOV     START_ADDR,DI   ;Save starting address in memory
                MOV     SI,ES:[DI]      ;Fetch index
EVEN_IDX:       TEST    SI,1            ;Force index to an even value
                JZ      ADD_IDX
                INC     SI
ADD_IDX:        ADD     DI,SI           ;Calculate next search address
COMPARE:        CMP     AX,ES:[DI]      ;Search value found?
                JE      ALL_DONE        ; If so, exit
                JA      HIGHER          ; Otherwise, find correct half
;
;  Search lower in the list.
;
                CMP     SI,2            ;Index = 2?
                JNE     IDX_OK
NO_MATCH:       STC                     ; If so, set CF
                JE      ALL_DONE        ;  and exit
IDX_OK:         SHR     SI,1            ; If not, divide index by 2
                TEST    SI,1            ;Force index to an even value
                JZ      SUB_IDX
                INC     SI
SUB_IDX:        SUB     DI,SI           ;Calculate next address
                JMP     SHORT COMPARE   ;Check this element
;
;  Search higher in the list.
;
HIGHER:         CMP     SI,2            ;Index = 2?
                JE      NO_MATCH        ; If so, go set CF and exit
                SHR     SI,1            ; If not, divide index by 2
                JMP     SHORT EVEN_IDX  ;  and go check next element
;
```

```
;  These are the exit instructions.
;
ALL_DONE:  MOV   SI,DI              ;Move compare address into SI
           MOV   DI,START_ADDR      ;Restore starting address
           RET                      ; and exit
B_SEARCH:  ENDP
```

The B_SEARCH procedure begins with a step that is not included in the basic binary search algorithm (Figure 5-4): it compares the search value with the first and last elements in the list. If the search value is less than the first element or greater than the last element, or if it matches one of those elements, the procedure terminates without further ado. If these initial checks fail, however, the 8088 proceeds to the search operation, starting at SEARCH.

After saving DI in memory, the 8088 copies the index (word count) from the first location of the list into SI, and forces it to an even value, if appropriate. This index is added to DI to form the address of the middle element in the list, the starting point for the binary search. The 8088 then compares the search value to this middle element, and jumps to ALL_DONE if the values match. In the absence of a match, the 8088 determines whether to continue its search in the upper half of the list (using the instructions that start at HIGHER) or the lower half.

These paths are similar in that both do the following:

1. Check whether the index is equal to 2. If it is, the 8088 sets CF to 1 (to indicate a non-match), then transfers to ALL_DONE to exit.
2. Divide the index by 2 shifting it right one position.
3. Force the shifted index to an even value.

However, to search lower in the list, the 8088 subtracts the index (SI) from the current address (DI); to search higher, it adds the index to the current address.

This process repeats until the index has decreased to 2 or the search value is found. Either way, the procedure ends at ALL_DONE, where DI is moved to SI and the original contents of DI are retrieved from memory.

How much more efficient is a binary search than a straight sequential, element-by-element scan—the kind we used in Examples 5-1 and 5-2? In "An Introduction to Algorithm Design" (*Computer*, February 1979, pp. 66-78), Jon L. Bentley stated that while a sequential search of an N-element list requires an average of N/2 comparisons, a binary search requires an average of $\log_2 N$ comparisons. Therefore, sequentially scanning a 100-element list will average 50 comparisons, but a binary search will do the same job with about seven comparisons!

Of course, you normally search a list for a value in order to *do* something to the matching element. Typically, you either want to add the value to the list or delete it from the list. Let's see how to perform those operations.

Adding an Element to an Ordered List

You add an element to an ordered list with four basic steps:

1. Find out where the value should be added.
2. Clear a location for the entry by moving all higher-valued elements down one position.
3. Insert the entry at the newly-vacated element position.
4. Add 1 to the list's element count, to reflect the insertion.

The B_SEARCH procedure we just developed gives us a good clue as to where the entry should be added, in that it returns the address of the element where the search stopped. To complete Step 1, we need to determine whether to insert the entry just *before* or just *after* that final search element. You can determine that by comparing the entry value to the final search element.

Example 5-7 shows a procedure (ADD_TO_OL) that performs the four steps we just listed. It begins by calling B_SEARCH, to find out whether the entry value is already in the list. Recall that B_SEARCH returns an address in SI and a found/not-found indicator in the Carry Flag (CF).

Example 5-7. Adding an element to an ordered list.

```
;  This procedure adds the value in AX to an ordered list
;  in the extra segment, if it is not already in the list.
;  The starting address of the list is in DI. The length
;  of the list, in words, is in its first location.
;  AX and DI are unaffected.
;  The B_SEARCH procedure (Example 5-6) is used to conduct
;  the search.
;
EXTRN       B_SEARCH:FAR
ADD_TO_OL   PROC
            PUSH   SI
            PUSH   CX
            PUSH   BX
            CALL   B_SEARCH     ;Is the value in the list?
            JNC    GOODBYE      ; Yes.  Exit
            MOV    BX,SI        ; No.  Copy compare addr. to BX
```

```
        MOV     CX,ES:[DI]  ;Find address of last element
        SHL     CX,1
        ADD     CX,DI       ;  and put it in CX
        PUSH    CX          ;Save end address on the stack
        SUB     CX,SI       ;Calculate no. of words to be moved
        SHR     CX,1
        CMP     AX,ES:[SI]  ;Should compare el. be moved, too?
        JA      EXCLUDE
        INC     CX          ; Yes.  Increase move count by 1
        JNZ     CHECK_CNT
EXCLUDE:    ADD     BX,2        ; No.  Adjust insert pointer
CHECK_CNT:  CMP     CX,0        ;Move count = 0?
        JNE     MOVE_ELS
        POP     SI          ; If so, store value at end of list,
        MOV     ES:[SI+2],AX
        JMP     SHORT INC_CNT ; then go increase element count
MOVE_ELS:   POP     SI          ;Start move at end of list
        PUSH    BX          ;Save insert address on stack
MOVE_ONE:   MOV     BX,ES:[SI]  ;Move one element down in list
        MOV     ES:[SI+2],BX
        SUB     SI,2        ;Point to next element
        LOOP    MOVE_ONE    ;Repeat until all are moved
        POP     BX          ;Retrieve insert address
        MOV     ES:[BX],AX  ;Insert AX in the list
INC_CNT:    INC     WORD PTR ES:[DI] ;Add 1 to element count
GOODBYE:    POP     BX          ;Restore registers
        POP     CX
        POP     SI
        RET                 ; and exit
ADD_TO_OL   ENDP
```

Upon return from B_SEARCH, the ADD_TO_OL procedure interrogates CF, and exits if CF is 0 (since that means the entry is already in the list). If CF is 1, however, the procedure saves the last-searched address in BX and calculates the address of the last element (in CX). Subtracting the contents of SI from this address gives the number of bytes that must be moved higher in memory to make room for the insertion. Right-shifting this result (that is, dividing it by 2) tells you how many *words* to move. If the entry value is less than the last-compared element, that element must also be moved, so the move count (CX) is increased by 1.

At CHECK_CNT, the 8088 checks the move count. If it is zero, the entry is simply tacked on to the end of the list. Otherwise, this value must be inserted in the list, which requires moving all subsequent elements down one word position.

The instructions starting at MOVE_ELS move elements down one by one, starting with the last word in the list. When all required elements have been moved, the 8088 inserts the entry (AX) in the newly-vacated slot, then increases the element count by 1.

Deleting an Element from an Ordered List

It is much easier to delete an element from an ordered list than it is to add one. All we must do is find the proper element, move all subsequent elements up one location, and decrement the element count.

Example 5-8 shows a typical delete procedure (DEL_OL) that uses B_SEARCH (Example 5-6) to locate the intended deletion "victim." As usual, the list's starting address is in DI and the value to be deleted is in AX.

If B_SEARCH locates the entry value in the list, DEL_OL uses its address, and the address of the end of the list, to find out how many words must be moved up in memory. The four-instruction loop at MOVEM performs the move operation. When the 8088 has moved all the words, it decreases the list's element count to reflect the deletion.

Example 5-8. Deleting an element from an ordered list.

```
;  This procedure deletes the value in AX from an ordered list
;  in the extra segment, if it is in the list.
;  The starting address of the list is in DI. The length
;  of the list, in words, is in its first location.
;  AX and DI are unaffected.
;  The B_SEARCH procedure (Example 5-6) is used to conduct
;  the search.
;
EXTRN   B_SEARCH:FAR
DEL_OL  PROC
        PUSH  SI
        PUSH  CX
        PUSH  BX
        CALL  B_SEARCH          ;Is the value in the list?
        JC    ADIOS             ; No.  Exit
        MOV   CX,ES:[DI]        ; Yes.  Find address of last element
        SHL   CX,1
        ADD   CX,DI             ;  and put it in CX
        CMP   CX,SI             ;Is the last element to be deleted?
        JE    CNT_M1            ; Yes.  Decrement element count
        SUB   CX,SI             ; No.  Calculate move count
        SHR   CX,1
```

```
MOVEM:  MOV   BX,ES:[SI+2]  ;Move one element up in list
        MOV   ES:[SI],BX
        ADD   SI,2          ;Point to next element
        LOOP  MOVEM         ;Repeat until all are moved
CNT_M1: DEC   WORD PTR ES:[DI]  ;Decrease element count by 1
ADIOS:  POP   BX            ;Restore registers
        POP   CX
        POP   SI
        RET                 ; and exit
DEL_OL  ENDP
```

5.4 Look-Up Tables

Many programs use tables to hold values that they need for processing. Sometimes these tables hold numbers that take too much time to calculate, such as sines of angles. Other times, they hold parameters that have some defined relationship to a program input, but which cannot be calculated. For instance, you can't give the computer someone's name and expect it to calculate his or her telephone number.

Applications like these call for a *look-up table.* As the name implies, a look-up table obtains an item of information (an *argument)* based on a known value (a *function*).

Look-up tables can replace complicated or time-consuming conversion operations, such as extracting the square root or cube root of a number, or deriving a trigonometric function (sine, cosine, etc.) of an angle. Look-up tables are especially efficient when a function covers only a small range of arguments (e.g., the cubes of numbers between 1 and 20). By using a look-up table, the computer doesn't need to perform complex calculations each time it needs a value.

In general, look-up tables save time in all but the simplest cases. (For instance, you wouldn't use a look-up table to store arguments that are always twice the value of a function.) Since look-up tables usually take up large amounts of memory storage space, they are most efficient in applications where you are willing to sacrifice memory space for execution speed.

Because look-up applications are so common, the 8088 has a special instruction, *Translate (XLAT),* for this purpose. XLAT looks up a value in a byte table, using the contents of BX as a base address and the contents of AL as an index into the table, and returns it in AL. This section includes examples of look-up operations on both byte tables and word tables.

Look-Up Tables Can Replace Equations

You can save processing time and program development time by providing the results of complex equations in a look-up table. As typical examples,

we will examine how look-up tables can provide the sine or cosine of an angle.

Sine of an Angle

As you may recall from high school trigonometry, the sines of all angles between 0° and 360° form the curve shown in Figure 5-5(A). Mathematically, you can approximate the curve with this formula:

$$\text{sine}(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} \ldots$$

It is possible to write a program to make this calculation, but it will probably take a few milliseconds to execute. If your application requires very precise sines, you may be forced to write such a program, but most applications are better served by an angle-to-sine look-up table.

If an application needs the sine of any angle between 0° and 360°, where the angle is an integer in degrees, how many sine values must the table contain? 360? No, we can get by with a table of only *91 sine values*—one for each angle between 0° and 90°.

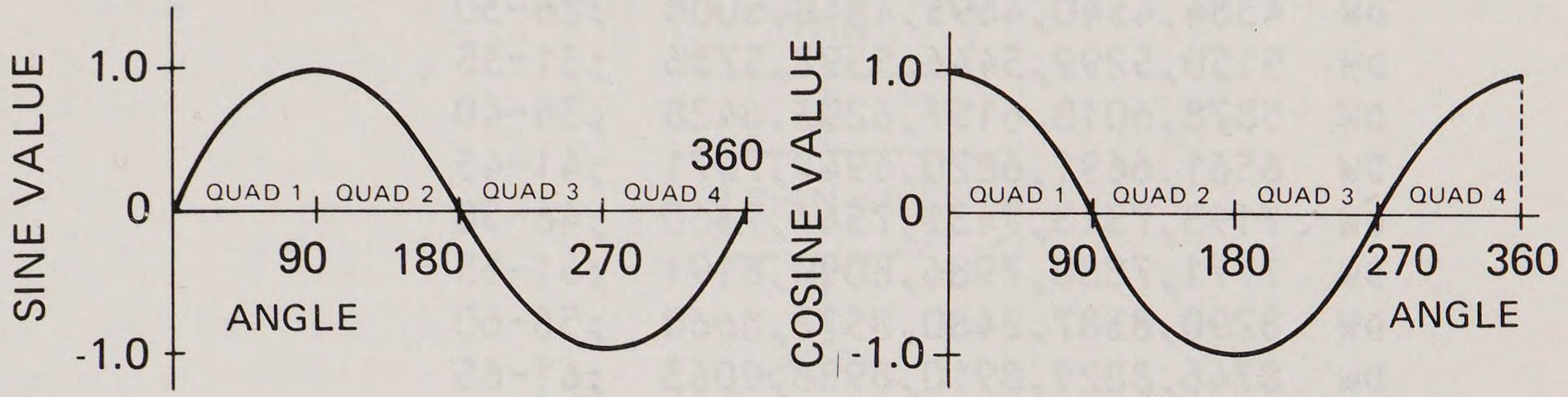
To understand how this can be so, look at Figure 5-5(A) again. If we call the leftmost quarter of the graph (angles from 0° to 90°) Quadrant I, we see that:

1. Sines in Quadrant II (91° to 180°) are the "mirror image" of those in Quadrant I.
2. Sines in Quadrant III (181° to 270°) are the "negative inverse" of those in Quadrant I.
3. Sines in Quadrant IV (271° to 360°) are the "negative inverse, mirror image" of those in Quadrant I.

That is, the sines in Quadrants II, III, and IV bear some simple relationship to those in Quadrant I.

Therefore, if you store the values in Quadrant I in a look-up table, your program can find the sine of an angle in any quadrant by making these conversions:

If the angle X is between	Take
0° and 90°	Sine(X)
91° and 180°	Sine(180 - X)
181° and 270°	-Sine(X - 180)
271° and 360°	-Sine(360 - X)



(A) SINES (B) COSINES

Figure 5-5. The sines and cosines of angles between 0 and 360 degrees.

These relationships let us construct a flowchart for an angle-to-sine conversion program. This flowchart, shown in Figure 5-6, derives the sine as a sign-and-magnitude value. That is, the high-order bit of the result gives the sign of the sine (0 = positive, 1 = negative) and the remaining bits give the sine's magnitude, its absolute value.

Example 5-9 gives an angle-to-sine look-up procedure called *FIND_SINE* that accepts angles from 0° to 360° in AX and returns a 16-bit sign-and-magnitude sine value in BX. The sines are stored as integers in the look-up table SINES. You must divide them by 10,000 to use them as operands.

To begin, the 8088 checks the size of the angle. If it is less than 181°, the processor jumps to SIN_POS; otherwise, it sets the most-significant bit of a sign register (CX) to 1—because sines above 180° are negative—and subtracts 180 from the angle.

Example 5-9. Look up the sine of an angle.

```
;  This procedure returns the sine of the angle (0 to 360
;  degrees) contained in AX.
;  The sine, a sign-and-magnitude value, is returned in BX.
;  The contents of AX are unaffected.
;
;  Store this sine look-up table in the data segment.
;
SINES   DW   0,175,349,523,698,872      ;0-5
        DW   1045,1219,1392,1564,1736   ;6-10
        DW   1908,2079,2250,2419,2588   ;11-15
        DW   2756,2924,3090,3256,3420   ;16-20
        DW   3584,3746,3907,4067,4226   ;21-25
```

```
        DW   4384,4540,4695,4848,5000   ;26-30
        DW   5150,5299,5446,5592,5736   ;31-35
        DW   5878,6018,6157,6293,6428   ;36-40
        DW   6561,6691,6820,6947,7071   ;41-45
        DW   7193,7313,7431,7547,7660   ;46-50
        DW   7771,7880,7986,8090,8191   ;51-55
        DW   8290,8387,8480,8572,8660   ;56-60
        DW   8746,8829,8910,8988,9063   ;61-65
        DW   9135,9205,9272,9336,9397   ;66-70
        DW   9455,9511,9563,9613,9659   ;71-75
        DW   9703,9744,9781,9816,9848   ;76-80
        DW   9877,9903,9926,9945,9962   ;81-85
        DW   9976,9986,9994,9998,10000   ;86-90
;
;  This is the look-up procedure.
;
FIND_SINE  PROC
           PUSH  AX                ;Save AX
           PUSH  CX                ; and CX
           SUB   CX,CX             ;Initialize sign to 0
           CMP   AX,181            ;Angle < 181 degrees?
           JB    SIN_POS           ; Yes.  Continue with sign = 0
           MOV   CX,8000H          ; No.  Set sign = 1
           SUB   AX,180            ;  and subtract 180 from angle
SIN_POS:   CMP   AX,91             ;Angle < 91 degrees?
           JB    GET_SIN           ; Yes. Go look up sine
           NEG   AX                ; No.  Subtract angle from 180
           ADD   AX,180
GET_SIN:   MOV   BX,AX             ;Make angle a word index
           SHL   BX,1
           MOV   BX,SINES[BX]      ; and look up the sine value
           OR    BX,CX             ;Combine sine with sign bit
           POP   CX
           POP   AX
           RET
FIND_SINE  ENDP
```

With the correct sign now in bit 15 of CX, the CMP instruction at SIN_POS compares the input angle to 91°. If the angle is greater than or equal to 91°, it must be subtracted from 180. You might expect to perform this subtraction with the instruction SUB 180,AX, but the 8088 does not offer this form. That being the case, we make the subtraction by negating AX, then adding 180 to the result. The four instructions at GET_SIN load the angle into BX, double it to form a word index, look up the sine in SINES, and OR it with the sign in CX.

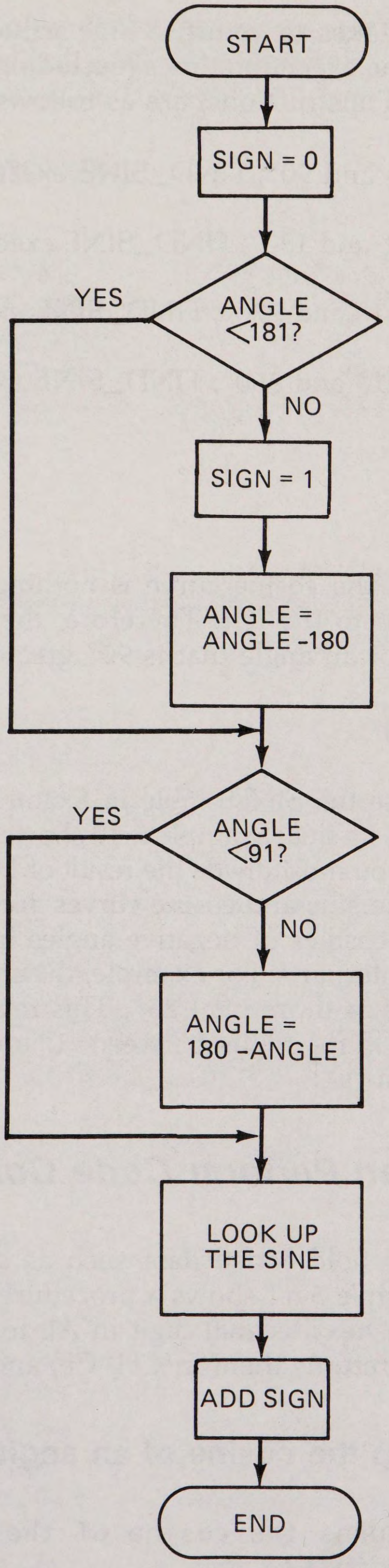


Figure 5-6. Flowchart for angle-to-sine conversion program.

How long FIND_SINE takes to return a sine value depends on which quadrant the angle is in. The execution times (excluding the time required to execute the CALL and RET instructions) are as follows:

- For angles between 0° and 90°, FIND_SINE executes in 125 cycles, or 26.25 μs.
- For angles between 91° and 180°, FIND_SINE executes in 120 cycles, or 25.20 μs.
- For angles between 181° and 270°, FIND_SINE executes in 121 cycles, or 25.41 μs.
- For angles between 271° and 360°, FIND_SINE executes in 116 cycles, or 24.36 μs.

Cosine of an Angle

As Figure 5-5(B) shows, the cosine curve is nothing more than the sine curve shifted one quadrant to the left. Therefore, the cosine of any given angle is equal to the sine of an angle that is 90° greater. That is,

$$\text{cosine}(X) = \text{sine}(X+90)$$

Knowing this, we can use the SINES table in Example 5-9 to look up the cosine of an angle as well as its sine. Example 5-10 shows a procedure that does this. As with FIND_SINE, you must divide the result of FIND_COS by 10,000.

Incidentally, note that the sine and cosine curves are symmetric about the vertical axis, so sines and cosines of negative angles have the same magnitudes as their positive counterparts. For example, the sine and cosine of -25° have the same magnitudes as those of +25°. This means you can also use FIND_SINE and FIND_COS for angles between -1° and -360° by supplying the angle's *absolute* value in AX.

Look-Up Tables Can Perform Code Conversions

Look-up tables can also hold coded data such as display codes, printer codes, and messages. Example 5-11 shows a procedure that performs multiple look-ups. It converts a hexadecimal digit in AL to its ASCII, BCD, and EBCDIC equivalents, and returns them in CH, CL, and AH, respectively.

Example 5-10. Look up the cosine of an angle.

```
;  This procedure returns the cosine of the angle (0 to 360
;  degrees) contained in AX.
```

```
;  The cosine, a sign-and-magnitude value, is returned in BX.
;  The contents of AX are unaffected.
;  This procedure calls FIND_SINE (Example 5-9).
;
EXTRN     FIND_SINE:FAR
FIND_COS  PROC
          PUSH  AX            ;Save AX
          ADD   AX,90         ;Add 90 for use by FIND_SINE
          CMP   AX,360        ;Is the result greater than 360?
          JNA   GET_COS
          SUB   AX,360        ; If so, subtract 360
GET_COS:  CALL  FIND_SINE     ;Look-up the cosine
          POP   AX
          RET
FIND_COS  ENDP
```

Example 5-11. Convert a hex digit to ASCII, BCD, and EBCDIC.

```
;  This procedure converts a hexadecimal digit in AL to its
;  ASCII, BCD, and EBCDIC equivalents.  The converted values
;  are returned in CH, CL and AH, respectively.
;  The contents of AL are unaffected.
;
;  Store these look-up tables in the data segment.
;
ASCII   DB  '0123456789ABCDEF'
BCD     DB  0,1,2,3,4,5,6,7,8,9,10H,11H,12H,13H,14H,15H
EBCDIC  DB  0F0H,0F1H,0F2H,0F3H,0F4H,0F5H,0F6H,0F7H
        DB  0F8H,0F9H,0C1H,0C2H,0C3H,0C4H,0C5H,0C6H
;
;  Here is the conversion procedure.
;
CONV_HEX  PROC
          PUSH  BX            ;Save BX and DX
          PUSH  DX
          MOV   DL,AL         ;Save the input value in DL
          LEA   BX,ASCII      ;Look-up the ASCII value
          XLAT  ASCII
          MOV   CH,AL         ; and load it into CH
          MOV   AL,DL
          LEA   BX,BCD        ;Look-up the BCD value
          XLAT  BCD
          MOV   CL,AL         ; and load it into CL
```

```
        MOV   AL,DL
        LEA   BX,EBCDIC ;Look-up the EBCDIC value
        XLAT  EBCDIC
        MOV   AH,AL     ; and load it into AH
        MOV   AL,DL     ;Restore registers
        POP   DX
        POP   BX
        RET
CONV_HEX  ENDP
```

When you transmit data between the computer and a printer, display, or some other peripheral in the system, it takes a form called *ASCII* (for American Standard Code for Information Interchange). *EBCDIC* (Extended Binary-Coded Decimal Interchange Code) is a transmission protocol for data processing and communications systems.

Jump Tables

Some look-up tables contain *addresses* rather than data values. For example, an error routine may use a look-up table to find the starting address of one particular message in a set of possible messages. Similarly, an interrupt service routine may use a look-up table to call one of several interrupt handler programs, depending on which type of service a particular device requested. Another routine may use a look-up table to call one of several subprograms, based on which option a user has selected from a menu. In all these applications (and there are many more), the look-up table that holds the addresses is called a *jump table.*

Example 5-12 illustrates a jump table application that might appear in a computer-assisted education program. This procedure, *SEL_OPT,* transfers to one of four routines (ADDITION, SUBTRACTION, MULTIPLICATION, or DIVISION) based on whether the student has selected 0, 1, 2, or 3 from a menu.

Example 5-12. Multiple-choice selection procedure.

```
;  This procedure calls one of four routines based on a
;  user selection code in AL.
;  The contents of AX and DI are affected.
;
;  This offset table must be stored in the data segment.
;
CHOICE   DW   ADDITION,SUBTRACTION,MULTIPLICATION,DIVISION
;
```

```
;  Here is the selection procedure.
;
SEL_OPT  PROC   FAR
         CMP    AL,3          ;Invalid choice?
         JA     ERROR
         CBW                  ; No.  Convert code to a word,
         MOV    DI,AX         ;  then move code into DI
         SHL    DI,1          ;  and convert it to an index
         JMP    CHOICE[DI]  ;Jump to routine
ERROR:   ..
         ..
         RET
ADDITION:
         ..
         ..
         RET
SUBTRACTION:
         ..
         ..
         RET
MULTIPLICATION:
         ..
         ..
         RET
DIVISION:
         ..
         ..
         RET
SEL_OPT  ENDP
```

SEL_OPT checks whether the entered code is valid, and jumps to an error-reporting routine if it is greater than 3. A valid code becomes an index that SEL_OPT uses to make an indirect jump to the chosen routine. When the student finishes, the routine's RET instruction returns control to the program that called SEL_OPT—the one that displays the menu.

5.5 Text Files

In the preceding sections we have been working with data structures that contain numbers. However, word processing and other applications involve manipulating non-numeric information—primarily text files.

Text files are lists whose elements are strings of ASCII characters. For instance, a text file that holds personnel information for a corporation will con-

tain one element, or *record,* for each employee. In turn, each record has several *fields* that list the employee's name, identification number, shift, pay rate, and so on. The string instructions are particularly convenient for manipulating text files.

You can manipulate text files with the same basic techniques you used for numeric files. But because text files have multi-character entries, the programs that operate on them must be somewhat different that those that operate on simple lists of bytes or words.

For instance, searching a text file usually requires you to compare a *part* of each entry (perhaps just the name field) to the search value, rather than comparing the entire entry. Similarly, bubble-sorting a text file requires you to compare single fields of adjacent entries, but move *entire* entries whenever an exchange is required.

As a simple example of a text file operation, let's look at a program that sorts a list of names and telephone numbers. The first location in the list holds a two-byte count of the entries.

Each entry in the list is 42 bytes long, divided into three fields: a 15-byte surname, a 15-byte first name/initial, and a 12-byte phone number. Any unused bytes in a field are assumed to contain ASCII "blank" characters. Hence, a typical entry in the list looks like this:

```
DB   'CORNELL',8 DUP (' ')
DB   'RAY',12 DUP (' ')
DB   '728-732-8437'
```

(Of course, you normally build text files by typing them in from the keyboard. For now, let's just assume the files are already in memory.)

Example 5-13 shows a procedure (PHONE_NOS) that bubble-sorts a telephone list stored in the extra segment. Its construction is similar to that of the "better bubble sort" procedure in Example 5-5.

Example 5-13. Sort a telephone list.

```
;  This procedure sorts a telephone list alphabetically.  The
;  list contains a word that holds a count of the entries,
;  followed by the entries.
;  Each entry is 42 bytes long, and divided into three fields:
;  a 15-byte surname, a 15-byte first name/initial, and a
;  12-byte phone number.
;  The list is in the extra segment, and its starting address
;  (the address of the count word) is in DI.
;
;  Set up these variables in the data segment.
;
```

```
FIRST_ENT    DW  ?
SAVE_CNT     DW  ?
;
;  Here is the main procedure.
;
PHONE_NOS  PROC
           PUSH  AX                  ;Save scratch registers
           PUSH  BX
           PUSH  CX
           PUSH  BP
           PUSH  DI
           PUSH  SI
           PUSH  DS
           CLD                       ;Set (DF) = 0, to move forward
           MOV   CX,ES:[DI]          ;Fetch entry count
           MOV   SAVE_CNT,CX         ; and save it in memory
           ADD   DI,2                ;Get address of first entry
           MOV   FIRST_ENT,DI        ; and save it in memory
INIT:      MOV   BX,1                ;Exchange flag (BX) = 1
           DEC   SAVE_CNT            ;Get ready for count-1 compares
           JZ    SORTED              ;Exit if SAVE_CNT is 0
           MOV   CX,SAVE_CNT         ;Load compare count into CX
           MOV   BP,FIRST_ENT        ; and first entry addr. into BP
NEXT:      MOV   DI,BP               ;Point to one entry with DI
           MOV   SI,BP               ; and the next with SI
           ADD   SI,42
           PUSH  CX                  ;Save current compare count
           MOV   CX,15               ;Compare 15-byte surname fields
                                     ;Is next surname < this surname?
REPE       CMPS  ES:BYTE PTR[SI], ES:[DI]
           JAE   CONT                ; No.  Check next pair
           MOV   CX,42               ; Yes.  Exchange these entries
SWAPEM:    MOV   AL,ES:[BP]
           XCHG  ES:[BP+42],AL
           MOV   ES:[BP],AL
           INC   BP
           LOOP  SWAPEM
           SUB   BX,BX               ;Set exchange flag = 0
CONT:      POP   CX                  ;Reload compare count
           LOOP  NEXT                ; and compare next two names
           CMP   BX,0                ;Were any exchanges made?
           JE    INIT                ;  If so, make another pass
SORTED:    POP   DS                  ;  If not, restore registers
           POP   SI
           POP   DI
```

```
            POP   BP
            POP   CX
            POP   BX
            POP   AX
            RET                        ;  and exit
PHONE_NOS   ENDP
```

After saving affected registers on the stack, the procedure reads the entry count and the address of the first entry into two variables, SAVE_CNT and FIRST_ENT. The instructions between NEXT and SWAPEM set up and execute a CMPS operation. Here, DI points to an entry's surname field and SI points to the next entry's surname field, where these fields lie 42 bytes apart in memory.

The loop at SWAPEM exchanges two entries when needed. Since the entries are 42 bytes long, loop counter CX is initialized to 42. The remainder of the procedure is similar to Example 5-5.

Note that PHONE_NOS does not account for duplicate surnames. Upon finding one, it *should* look at the first name fields of those entries and perform a second, alphabetical sort to put them in order. You might like to modify Example 5-13 to do this.

Study Exercises (answers on page 364)

1. The list-processing procedures in this chapter don't check whether the list is empty (it has a count—which contains 0—but no data elements) before conducting an add, delete, or search operation. To remedy this, modify Example 5-1 so that AX becomes the first data element in a list if the list is empty. You can then use the modified procedure to build *new* lists as well as to add elements to existing lists.
2. Write a procedure that searches an ordered list for the value in AX and, if the value is found, *replaces* the contents of the matching element with the value in BX.

6

Using the System Resources

The preceding chapters of this book deal mostly with the 8088 microprocessor and, therefore, apply to any 8088-based microcomputer. In this chapter you will learn how to use the built-in resources of your IBM PC or XT.

By built-in resources, we mean the programs that form the computer's main operating program; its Basic I/O System or *BIOS*. The BIOS stores characters typed from the keyboard, displays characters on the screen, and transfers data to and from the disk drives, printer, and whatever else is attached to your computer.

In short, without BIOS, the PC would be just a collection of metal, plastic, and electrical circuits. BIOS is the PC's native intelligence that makes it a computer.

We will first take a brief look at how memory is organized in the PC and XT, then we will describe usable programs that are built into the BIOS and other programs that DOS puts into memory. Most of these are *interrupt* service routines, which means your programs can call them by executing appropriate INT instructions.

6.1 System Memory

Figure 6-1 shows how the 8088's one-megabyte address space is organized in the PC and XT. The computer's main *System Board* comes with a minimum 64K bytes of Read/Write (RAM) memory, which responds to addresses 0 through 0FFFF. As the "blow-up" on the right shows, not all of this space is available to users. The lowest 1.5K bytes hold interrupt vectors (addresses of interrupt service routines) and some data areas that the BIOS, BASIC, and DOS use for working storage.

The System Board can accommodate 192K additional bytes (locations 10000 through 3FFFF), bringing the on-board total to 256K bytes. The 384K bytes starting at location 40000 accept add-on memory boards, which can bring the computer to its 640K limit.

Following the add-on area is a block of 16K locations that IBM reserves for some mysterious future use. Location A4000 is the start of a 112K block of memory where the PC stores graphics and display data.

Starting at location C0000, the rest of the memory space is dedicated to ROM usage. IBM calls the first 32K locations a "ROM expansion area," but only they know what they plan to do with it. The 8K bytes starting at location C0000 are where the ROM in the PC XT's fixed disk adapter resides. Following it is a block of 168K locations for ROMs on other adapter cards. The 8K bytes starting at location F4000 are available for a user-provided ROM chip, which you can plug into a spare socket on the System Board. Finally, the highest-numbered 40K locations, F6000 through FFFFF, hold the Cassette BASIC and BIOS ROM chips.

If you have the IBM Macro Assembler or some other relocating assembler, you needn't worry about where the computer stores your programs and data in memory. However, you *should* understand the interrupt assignments in the system, so you can use the built-in interrupts and add others of your own.

As the blow-up in Figure 6-1 shows, BIOS, DOS, and BASIC use interrupt Types 0 through 1FH, 20H through 3FH, and 80H through FFH (respectively), and IBM has reserved Types 40H through 5FH. This leaves 32 interrupts (Types 60H through 7FH) available for your use.

6.2 BIOS Interrupts

The BIOS interrupts give you a way to access the PC's powerful built-in features. Table 6-1 summarizes the BIOS interrupt vector assignments. For reference, we also show what segment number and offset each vector receives when you switch the power on. The segment number and offset are separated with a colon.

Don't use these addresses in your programs (they differ between PC models and versions of DOS); we give them only to help you find listings of interrupt service routines in the *Technical Reference* manual. Appendix A of the manual contains a complete listing of the BIOS, with the offset in the leftmost column (the segment number for the BIOS is F000, as the memory map in Figure 6-1 shows). To look up the PRINT_SCREEN routine for the Type 5 interrupt, for example, scan the listing until you reach *FF54*. This is somewhere around page A-80 (it varies between different editions and printings). Appendix A of the *PC/XT Technical Reference* manual has a separate listing for the fixed disk adapter's BIOS. This ROM starts at segment C800 (again, see Figure 6-1).

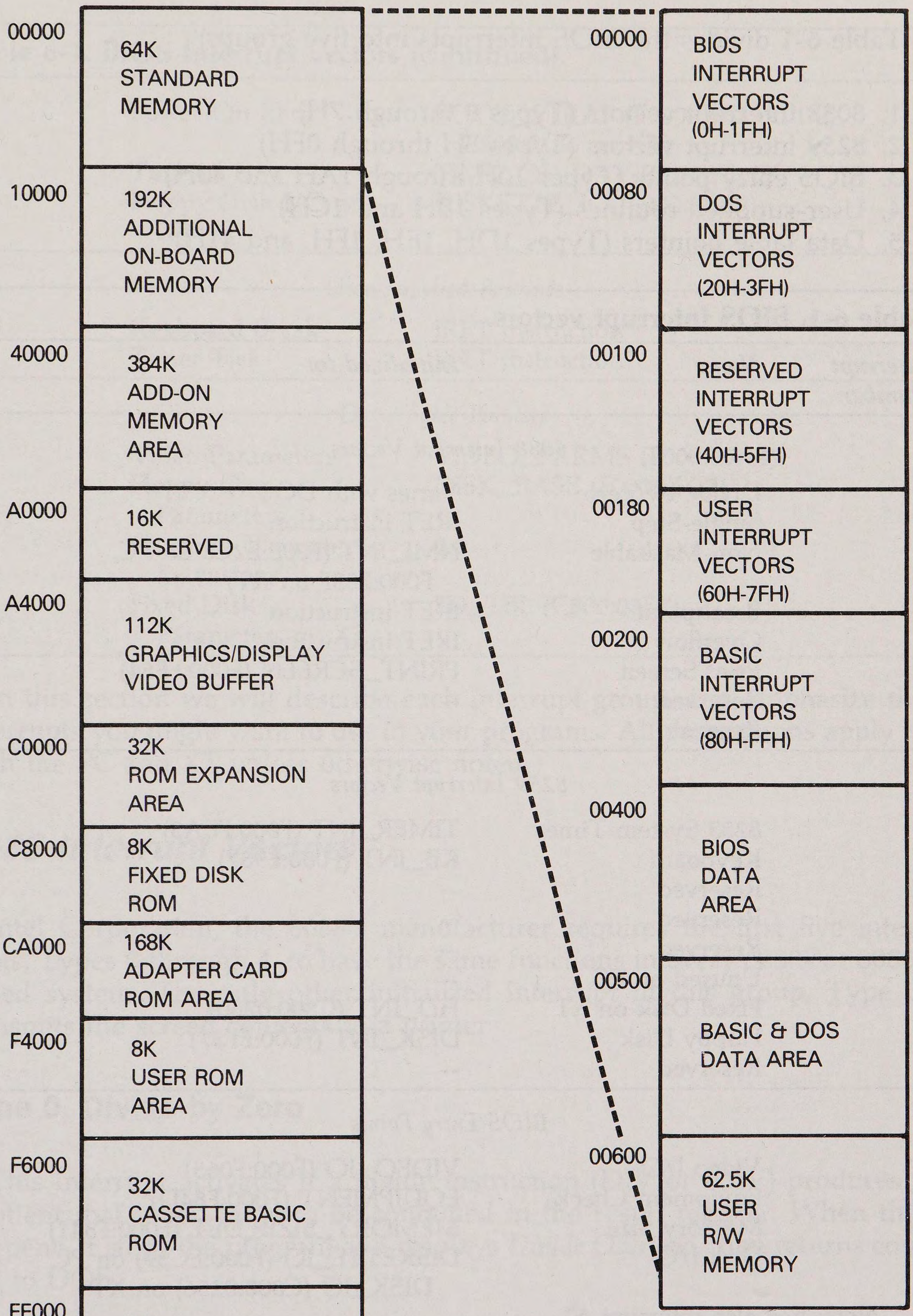


Figure 6-1. System memory map.

Table 6-1 divides the BIOS interrupts into five groups:

1. 8088 interrupt vectors (Types 0 through 7H)
2. 8259 interrupt vectors (Types 8H through 0FH)
3. BIOS entry points (Types 10H through 1AH and 40H)
4. User-supplied routines (Types 1BH and 1CH)
5. Data table pointers (Types 1DH, 1EH, 1FH, and 41H)

Table 6-1. BIOS interrupt vectors.

Interrupt Number	*Name*	*Initialized to:*
	8088 Interrupt Vectors	
0	Divide by Zero	Varies with DOS
1	Single-Step	IRET instruction
2	Non-Maskable	NMI_INT (F000:E2C3 on PC, F000:F85F on XT)
3	Breakpoint	IRET instruction
4	Overflow	IRET instruction
5	Print Screen	PRINT_SCREEN (F000:FF54)
6	Reserved	--
7	Reserved	--
	8259 Interrupt Vectors	
8	8253 System Timer	TIMER_INT (F000:FEA5)
9	Keyboard	KB_INT (F000:E987)
A	Reserved	--
B	Reserved	--
C	Reserved	--
D	Unused on PC	--
	Fixed Disk on XT	HD_INT (C800:0760)
E	Floppy Disk	DISK_INT (F000:EF57)
F	Reserved	--
	BIOS Entry Points	
10	Video I/O	VIDEO_IO (F000:F065)
11	Equipment Check	EQUIPMENT (F000:F84D)
12	Memory Size	MEMORY_SIZE_DET (F000:F841)
13	Disk I/O	DISKETTE_IO (F000:EC59) on PC DISK_IO (C800:0256) on XT
	Note: See also interrupt 40.	
14	Communications I/O	RS232_IO (F000:E739)
15	Cassette I/O	CASSETTE_IO (F000:F859)
16	Keyboard I/O	KEYBOARD_IO (F000:E82E)
17	Printer I/O	PRINTER_IO (F000:EFD2)
18	Cassette BASIC	(F600:0000)

Table 6-1. BIOS interrupt vectors (continued).

19	Power-On Reset	BOOT_STRAP (F000:E6F2 on PC, C800:0186 on XT)
1A	Time of Day	TIME_OF_DAY (F000:FE6E)
40	Floppy Disk I/O on XT	DISKETTE_IO (F000:EC59)
	User-Supplied Routines	
1B	Keyboard Break	IRET instruction
1C	Timer Tick	IRET instruction
	Data Table Pointers	
1D	Video Parameters	VIDEO_PARMS (F000:F0A4)
1E	Floppy Disk Parameters	DISK_BASE (F000:EFC7)
1F	Graphics Character Extensions	0:0
41	Fixed Disk Parameters on XT	FD_TBL (C800:03E7)

In this section we will describe each interrupt group, and emphasize the interrupts you might want to use in your programs. All descriptions apply to both the PC and XT unless otherwise noted.

8088 Interrupt Vectors

Intel Corporation, the 8088's manufacturer requires the first five interrupts, Types 0 through 4, to have the same functions in every 8088- or 8086-based system. The only other initialized interrupt in this group, Type 5, transmits the screen contents to a printer.

Type 0, Divide by Zero

This interrupt activates if a divide instruction (DIV or IDIV) produces a quotient that is too large to be contained in the result register. When that happens, it stops the program and displays *Divide Overflow*, then returns control to DOS.

Type 1, Single-Step

This interrupt is designed to let you run programs one instruction at a time, so you can debug them. DOS makes this vector point to an Interrupt

Return (IRET) instruction. Thus, an INT 1 instruction makes the 8088 jump to the IRET instruction, then return to the instruction that follows INT 1. IBM doesn't use this interrupt because they let you single-step with DEBUG's Trace (T) command.

Type 2, Non-Maskable Interrupt

For every interrupt but this one, you can force the processor to ignore interrupt requests. To do this, set the Interrupt Enable Flag (IF) to zero with a CLI instruction. Type 2 is a *non-maskable* interrupt; it cannot be "locked out" or disabled.

The PC's memory expansion options use the Non-Maskable Interrupt (NMI, for short) to report storage errors. The service routine this interrupt calls, NMI_INT, displays the error message *Parity Check 1* (for errors on the System Board) or *Parity Check 2* (for errors on add-on boards). Then it locks out interrupts with a CLI instruction and halts the processor with HLT.

If you have an 8087 Math Coprocessor chip installed, it uses the NMI interrupt to report errors. In this case, of course, you need additional software to determine whether the interrupt is due to a memory error or an 8087 error.

Type 3, Breakpoint

This interrupt lets you execute a program until the 8088 encounters a specified "stop" address, or *breakpoint*. DOS makes the Type 3 interrupt vector point to an IRET instruction, but DEBUG temporarily changes it if you use the breakpoint option of the Go (G) command.

Type 4, Overflow

This interrupt activates when the 8088 executes an Interrupt If Overflow (INTO) instruction. DOS makes the Type 4 interrupt vector point to an IRET instruction, because IBM can't anticipate what you want to do in case of overflow. They leave this decision up to you.

Type 5, Print Screen

The final interrupt in this group, Type 5, is a BIOS feature. The service routine this interrupt calls, PRINT_SCREEN, saves the current cursor position, transmits the information on the screen to the printer, then restores the cursor. Hence, the Type 5 interrupt does the same thing as pressing Shift-PrtSc, but under program control instead of from the keyboard.

Memory address 50:0 holds the print status, as follows:

- During printing, location 50:0 contains one.
- If the print operation is successful, location 50:0 contains 0.
- If an error occurs during printing, location 50:0 contains 0FFH.

8259 Interrupt Vectors

The 8259, an "interrupt controller" chip on the System Board, accepts interrupt request signals (that is, maskable interrupts) from up to eight different devices in the system. Acting like an electronic traffic officer, it passes the request signal and a device identifier code to the 8088. In the PC, only the 8253 system timer, keyboard, and floppy disk controller are attached to the 8259. In the XT, the fixed disk controller is also attached.

Type 8, 8253 System Timer

The 8253 is a chip on the System Board chip that maintains a count of the system clock. You can use it to calculate the elapsed time between two events or to generate time delays. The 8253 issues a Type 8 interrupt every 0.0549254 seconds; that is, it interrupts the 8088 about 18.2 times a second.

Type 8's service routine, TIMER_INT, keeps track of the 8253's interrupts while interrupts are enabled (that is, while IF = 1), so you can use this count to keep track of the time of day. The time count is a 32-bit value in two 16-bit memory locations, TIMER_LOW (0040:006C) and TIMER_HIGH (0040:006E). The TIMER_INT routine also issues a Type 1C interrupt at every "timer tick."

BIOS makes the Type 1C interrupt vector point to an IRET instruction, so you must change the vector if you want the interrupt to do something useful. We'll discuss some possibilities for the Type 1C interrupt under *User-Supplied Routines* in this section. We'll also show you how to set and read the time count when we discuss the Type 1A (Time of Day) interrupt under *BIOS Entry Points.*

Type 9, Keyboard

This interrupt activates when you press or release a key. For all practical purposes, consider Type 9 a system interrupt. We'll discuss a more useful keyboard interrupt, Type 16, under BIOS Entry Points.

Types D and E, Fixed Disk and Floppy Disk

These are the interrupts that BIOS receives when a disk needs to send or receive data. As with Type 9, consider Types D and E as system interrupts. We'll discuss a more useful disk interrupt, Type 13, under *BIOS Entry Points.*

BIOS Entry Points

Most of these interrupts perform input and output functions; they let you transfer information to or from the peripherals in the system. Other interrupts in this group let you determine the system configuration and the amount of memory installed, and initialize and read the time of day.

Type 10, Video I/O

This interrupt performs any of 16 different display operations based on a value in AH. Type 10's service routine, VIDEO_IO, begins by loading the starting address of the *display buffer* (a block of memory that holds screen characters) into the Extra Segment (ES) register. If your PC has a Color/Graphics Monitor Adapter card, the display buffer is 16K bytes long and starts at location B8000. If it has a Monochrome Display/Printer Adapter card, the buffer is 4K bytes long and starts at location B0000.

With ES properly initialized, VIDEO_IO performs the specified I/O operation. Table 6-2 summarizes these operations, and shows which registers they use. We divide the video operations into five groups:

- *CRT Interface Routines* set the video mode and the line limits for the cursor, set and read the cursor position, read the light pen position, and control the active display page.
- *Character-Handling Routines* transfer characters to and from the screen.
- *Graphics Interface Routines* display and read graphics dots, and change the colors.
- The *ASCII Teletype Routine* writes a character to the screen then advances the cursor.
- The *Read Video State Routine* reports the current mode, screen width, and display page.

You probably have questions about certain terms in Table 6-2 (perhaps "attribute" and "color value") and about the service routines themselves. Please be patient; we will (hopefully) answer them when we cover graphics in Chapter 7.

Table 6-2. Video I/O operations with Type 10 interrupt.

(AH)	*Operation*	*Additional Input Registers*	*Result Registers**
CRT Interface Routines			
0	Set video mode	(AL) = 0 40×25 B/W, Alpha (Default) = 1 40×25 Color, Alpha = 2 80×25 B/W, Alpha = 3 80×25 Color, Alpha = 4 320×200 Color, Graphics = 5 320×200 B/W, Graphics = 6 640×200 B/W, Graphics	None
1	Set cursor lines	CH Bits 0-4 = Start line for cursor CH Bits 5-7 = 0 CL Bits 0-4 = End line for cursor CL Bits 5-7 = 0	None
2	Set cursor position	(DH,DL) = Row, column (0,0) is upper left (BH) = Page number (0 for Graphics mode)	None
3	Read cursor position	(BH) = Page number (0 for Graphics mode)	(DH,DL) = Row,column of cursor (CH,CL) = Current cursor mode

Table 6-2. Video I/O operations with Type 10 interrupt (continued).

(AH)	*Operation*	*Additional Input Registers*	*Result Registers**
4	Read light pen position	None	(AH) = 0 Light pen switch not down or not triggered (AH) = 1 Valid light pen values in registers (DH,DL) = Row,column (CH) = Raster line (0-199) (BX) = Pixel column (0-319,639)
5	Select active display page (Alpha modes)	(AL) = New page value (0-7 for Modes 0 and 1; 0-3 for Modes 2 and 3)	None
6	Scroll active page up	(AL) = Number of lines, Input lines blanked at bottom of window. (AL) = 0 blanks entire window. (CH,CL) = Row,column of upper left corner of scroll (DH,DL) = Row,column of lower right corner of scroll (BH) = Attribute to be used on blank line	None

Table 6-2. Video I/O operations with Type 10 interrupt (continued).

(AH)	*Operation*	*Additional Input Registers*	*Result Registers**
7	Scroll active page down	(AL) = Number of lines, Input lines blanked at top of window. (AL) = 0 blanks entire window. (CH,CL) = Row,column of upper left corner of scroll (DH,DL) = Row,column of lower right corner of scroll (BH) = Attribute to be used on blank line	None
Character-Handling Routines			
8	Read attribute/ character at current cursor position	(BH) = Display page (Alpha modes)	(AL) = Character read (AH) = Attribute of character read (Alpha modes)
9	Write attribute/ character at current cursor position	(BH) = Display page (Alpha modes) (BL) = Attribute of character (Alpha) = Color of character (Graphics) (CX) = Count of characters to write (AL) = Character to write	None
10	Write character only at current cursor position	(BH) = Display page (Alpha modes) (CX) = Count of characters to write (AL) = Character to write	None
Graphics Interface			
11	Set color palette (320×200 graphics)	(BH) = ID of palette color (0-127) (BL) = Color value to be used with that color ID	None

Table 6-2. Video I/O operations with Type 10 interrupt (continued).

(AH)	*Operation*	*Additional Input Registers*	*Result Registers**
12	Write dot	(DX) = Row number (CX) = Column number (AL) = Color value If Bit 7 of AL = 1, the color value is exclusive-ORed with the current contents of the dot	None
13	Read dot	(DX) = Row number (CX) = Column number	(AL) = Dot read
ASCII Teletype Routine for Output			
14	Write character to screen, then advance cursor	(AL) = Character to write (BL) = Foreground color (Graphics) (BH) = Display page (Alpha)	None
Read Video State			
15	Read current video state	None	(AL) = Current mode—See (AH) = 0 for explanation (AH) = Number of character columns on screen (BH) = Current active display page

Note: Besides the registers listed here, these routines preserve CS, SS, DS, ES, BX, CX, and DX. All other registers should be considered destroyed.

Type 11, Equipment Check

The Type 11 interrupt determines what options are attached to the computer (based on the switch settings), and returns a summary in AX; see Figure 6-2. For example, on a PC/XT with a Monochrome Display and a printer, but no modem or game devices, Type 11 returns (AX) = 423D.

Of course, this information is irrelevant if you are writing software for your own computer, because you already know what equipment it has. However, it is valuable for developing commercial software. Using the contents of AX, you can customize your software to run on different systems. To do this, use the contents of AX to call one of several system-specific procedures.

Note especially that bit 1 of AX tells whether the computer has an 8087 Math Coprocessor. If you have written a program that can take advantage of the 8087, use Type 11 to tell it whether to either execute or bypass 8087 instructions.

Type 12, Memory Size

The Type 12 interrupt determines how many 1K blocks of RAM memory are installed (based on the switch settings), and returns this count in AX.

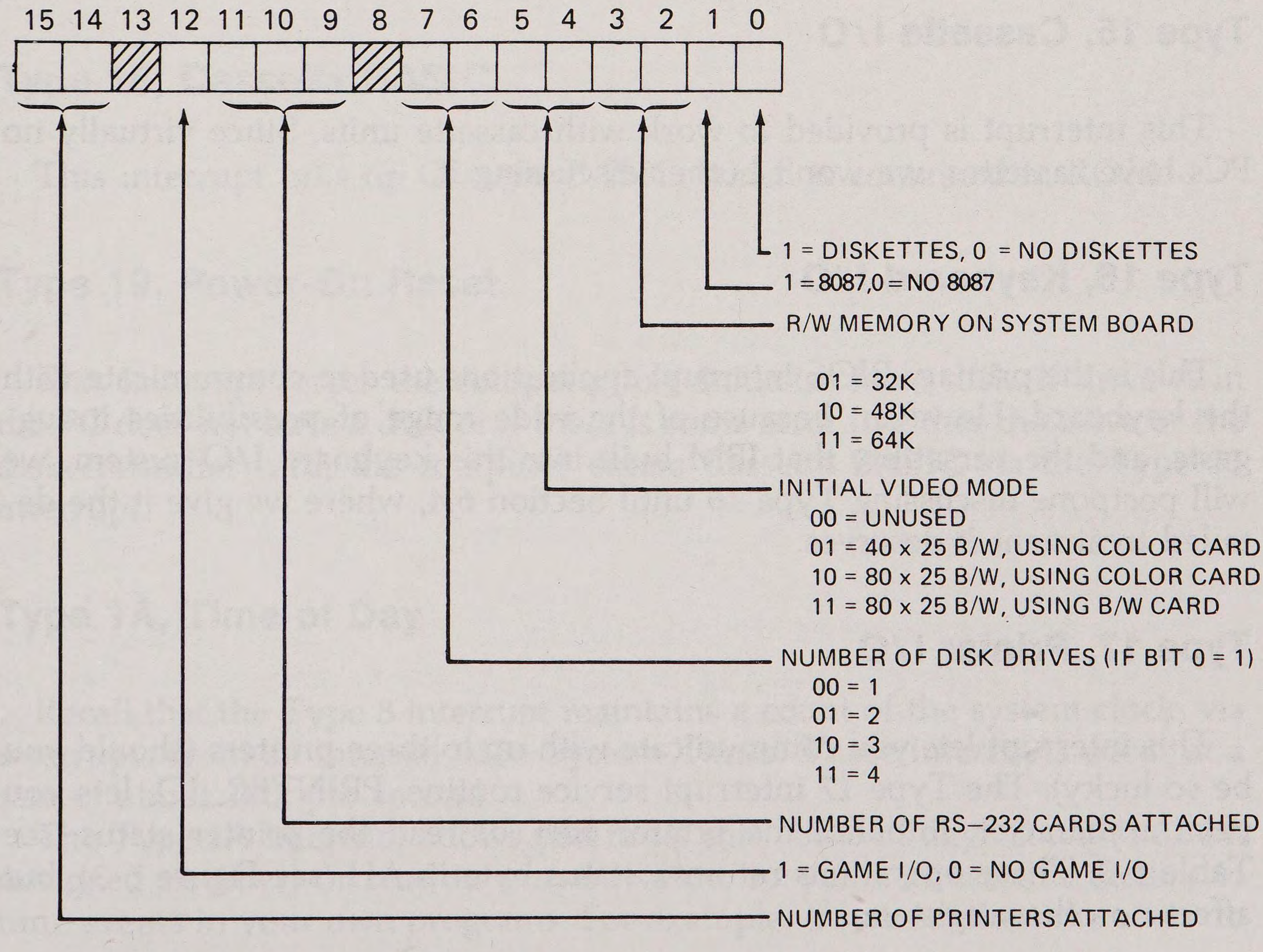


Figure 6-2. Equipment status returned in AX by interrupt 11.

Type 13, Disk I/O

The DOS commands let you operate on *files* in the system. The Type 13 interrupt gives you a greater degree of control over disk information in that it lets you operate on individual *tracks* or *sectors*. The PC's BIOS initializes the Type 13 interrupt vector to point to a floppy disk service routine called DISKETTE_IO; the XT's BIOS makes it point to a fixed disk routine called DISK_IO. (DISKETTE_IO is still in the XT's BIOS, but you must use Type 40 rather than Type 13 to access it.)

Type 13 provides many functions that are valuable for developing disk utilities or copy-protection schemes. But be warned: These operations are *not* for the faint-hearted. Before attempting them, you'd better know what you're doing! For guidance, consult the IBM *Technical Reference* manual.

Type 14, Communications I/O

This interrupt lets you transmit and receive information through the PC's serial communication port. For more information, see the description of the Asynchronous Communications Adapter in the IBM *Technical Reference* manual.

Type 15, Cassette I/O

This interrupt is provided to work with cassette units. Since virtually no PCs have cassettes, we won't bother discussing it.

Type 16, Keyboard I/O

This is the primary BIOS interrupt applications used to communicate with the keyboard. However, because of the wide range of possibilities it suggests, and the versatility that IBM built into this keyboard I/O system, we will postpone discussing Type 16 until Section 6.4, where we give it the detailed treatment it deserves.

Type 17, Printer I/O

This interrupt lets you communicate with up to three printers (should you be so lucky). The Type 17 interrupt service routine, PRINTER_IO, lets you print a character, initialize the printer port, or read the printer status; see Table 6-3. These operations return a status byte in AH (see Figure 6-3), but affect no other registers.

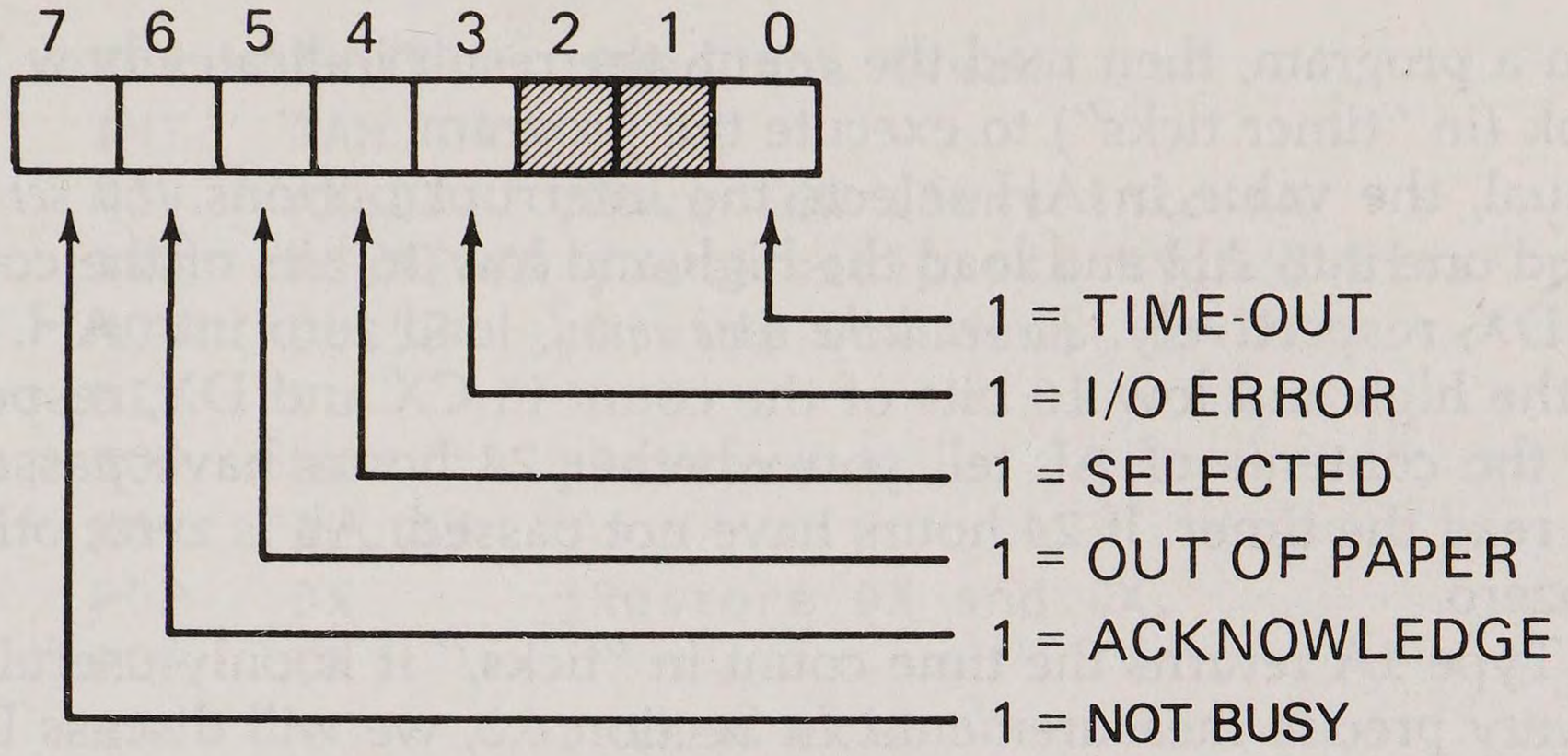


Figure 6-3. Printer I/O status byte.

Table 6-3. Printer I/O operations with the Type 17 interrupt.

(AH)	*Operation*	*Additional Input Registers*	*Result Registers*
0	Print a character	(AL) = Character (DX) = Printer (0-2)	(AH) = Status of operation (see Figure 6-3)
1	Initialize printer	(DX) = Printer (0-2)	Same as print routine
2	Read printer status	(DX) = Printer (0-2)	Same as print routine

Type 18, Cassette BASIC

This interrupt calls up Cassette BASIC, which is contained in ROM.

Type 19, Power-On Reset

This interrupt forces the computer to reinitialize from disk. If drive A in the PC does not have a disk or if there is some error in either the drive or the disk controller card, the computer enters Cassette BASIC via the Type 18 interrupt.

Type 1A, Time of Day

Recall that the Type 8 interrupt maintains a count of the system clock, via interrupts from an internal 8253 System Timer. These interrupts occur at a rate of about 18.2 per second.

The Type 1A interrupt allows you to set this "time of day" count (a 32-bit unsigned number) or read its current value, so you can use this facility to time events in your own programs. For example, if you set the time count to

zero, run a program, then read the count, the result indicates how long the 8088 took (in "timer ticks") to execute the program.

As usual, the value in AH selects the interrupt options. *To set the time count,* load one into AH and load the high and low 16 bits of the count into CX and DX, respectively. *To read the time count,* load zero into AH. The PC returns the high and low 16 bits of the count in CX and DX, respectively. Further, the contents of AL tell you whether 24 hours have passed since you last read the timer. If 24 hours have not passed, AL is zero; otherwise, it is nonzero.

Since Type 1A returns the time count in "ticks," it is only useful for obtaining very precise measurements. In Section 6.3, we will discuss DOS interrupts that let you set and read the time in hours, minutes, seconds, and hundredths of a second. Still, Type 1A can be useful for one thing: generating random numbers.

Generating Random Numbers

Because the time count locations TIMER_LOW (0040:006C) and TIMER_HIGH (0040:006E) continually change, their contents—particularly those of TIMER_LOW—can be interpreted as a *pseudo-random* number! Since the count locations always increase rather than change randomly, they don't provide a true random number. But because the time value increases 18.2 times per second, it is "random" enough for most applications.

Being 16 bits long, TIMER_LOW can hold values between 0 and 65,535. To generate a random number between 0 and 51, as you might want to do in a card game program, you could read TIMER_LOW and divide it by 52. This will produce a *remainder* between 0 and 51.

Example 6-1 shows a procedure called RAND_51 that returns a value between 0 and 51 in AH. Note that we strip off the three high bits of TIMER_LOW to prevent the division from overflowing. If we didn't do this, the division will overflow whenever TIMER_LOW exceeds decimal 13260 (i.e., 52 x 255).

Example 6-1. Generate a random number between 0 and 51.

```
;  This procedure uses TIMER_LOW from the Type 1A interrupt to
;  generate a pseudo-random number between 0 and 51.  The
;  number is returned in AH.  No other registers are affected.
;
RAND_51  PROC
         PUSH  CX        ;Save registers affected by INT 1AH
         PUSH  DX
         PUSH  AX
```

```
        MOV     AH,0        ;Read the timer
        INT     1AH
        MOV     AX,DX       ;Move low count into AX
        AND     AX,1FFFH ; and strip off high 3 bits
        MOV     DL,52       ;Divide low count by 52
        DIV     DL
        POP     DX          ;Restore AL
        MOV     AL,DL
        POP     DX          ;Restore DX and CX,
        POP     CX
        RET                 ; then exit
RAND_51 ENDP
```

User-Supplied Routines

These two interrupts let you add a couple of enhancements to the system.

Type 1B, Keyboard Break

This interrupt activates when you press Ctrl-Break on the keyboard. It is invoked when the 8088 is responding to the Keyboard interrupt, Type 9. BIOS makes the Type 1B interrupt point to an IRET instruction, but DOS changes it to point to the service routine for the Type 23 (Ctrl-Break Exit Address) interrupt. Type 23, a DOS interrupt, is described in Section 6.3.

Type 1C, Timer Tick

This interrupt is invoked by TIMER_INT, the routine that services the Type 8 (8253 System Timer) interrupt. Thus, like the Type 8 interrupt, Type 1C executes about 18.2 times per second while interrupts are enabled.

The Type 1C interrupt lets you give the 8088 some additional task with each "tick" of the 8253. BIOS makes the Type 1C interrupt vector point to an IRET instruction, so Type 1C does nothing unless you change this pointer.

You can use the Type 1C interrupt to display the time count in decimal to create a time-of-day clock. Or you could make Type 1C's service routine examine the time count, and beep the speaker when the time reaches a certain value; this gives you a computerized alarm clock. You can probably think of other applications for this useful interrupt.

Data Table Pointers

Types 1D through 1F, and 41, are not interrupt vectors at all; they are addresses of data tables. The 1D address points to a table used by the Type 10 interrupt. The 1E address points to a floppy disk table used by the Type 13 interrupt on a PC or the Type 40 interrupt on an XT. The 41 address points to a fixed disk table used by the Type 13 interrupt on an XT.

Type 1F's location is reserved for a pointer to a graphics table for special characters, but since this table is not in memory, BIOS initializes the vector to 0:0. The DOS 3.0 GRAFTABL.COM routine provides an extended character set. When loaded, it makes the 1F vector point to a 1K table that provides 128 additional characters for the graphics mode. Each character is defined in an 8 x 8 bit (or 8-byte) array, where the first byte defines the top row of the character.

6.3 DOS Interrupts

IBM reserves interrupt types 20 through 3F for DOS use, of which Types 20 through 27 are currently implemented. Table 6-4 summarizes these interrupts.

Table 6-4. DOS interrupts.

Interrupt Number	*Name*
20	Terminate Program
21	Function Calls
22	Terminate Address
23	Ctrl-Break Exit Address
24	Critical Error Handler
25	Absolute Disk Read
26	Absolute Disk Write
27	Terminate, But Stay Resident
28-3F	Reserved for DOS

Most of these interrupts are useful only to DOS, so we won't discuss them here. (If you have DOS 1.1 or 2.0, see Appendix D of the IBM *Disk Operating System* manual for details. If you have DOS 2.1 or above, see the separate *DOS Technical Reference* manual.) However, the Type 21 (function calls) interrupt has a variety of convenient options for interacting with the keyboard, display, printer, disk, and asynchronous communications device. Table 6-5 summarizes the most useful Type 21 function calls.

Function calls 0 through 2E are common to all versions of DOS after 1.0, while function calls 2F and higher require DOS 2.0 or later. Many of these

Table 6-5. Function calls with the Type 21 interrupt.

(AH) Operation	*Additional Input Registers*	*Result Registers*
Keyboard Functions		
1 Wait for keyboard character, then display it (with Ctrl-Break check)*	None	(AL) = Character
6 Read keyboard character (no Ctrl-Break check)*	(DL) = 0FFH	(AL) = Character, if available = 0, if no character is available
7 Wait for keyboard character, but do not display it (no Ctrl-Break check)*	None	(AL) = Character
8 Same as function 7, but with Ctrl-Break check*	None	(AL) = Character
A Read keyboard string into buffer	(DS:DX) = Buffer address First buffer byte = Buffer size	Second buffer byte = Number of chars. read
B Read keyboard status	None	(AL) = 0FFH if no character is available = 0 if character is available
C Clear keyboard buffer and call a keyboard function	(AL) = Keyboard function number (1, 6, 7, 8, or A)	Per keyboard function
Display Functions		
2 Display a character (with Ctrl-Break check)	(DL) = Character	None
5 Print a character	(DL) = Character	None
6 Display a character (no Ctrl-Break check)	(DL) = Character	None

*Some key combinations generate "extended codes," and may require two function calls. See Section 6.4 for details.

Table 6-5. Function calls with the Type 21 interrupt (continued).

(AH)	*Operation*	*Additional Input Registers*	*Result Registers*
9	Display a string	(DS:DX) = String address String must end with $.	None
Asynchronous Communications Functions			
3	Wait for asynchronous input character	None	(AL) = Character
4	Output a character to asynchronous device	(DL) = Character	None
File Management Functions			
D	Reset default disk drive	None	None
E	Select default disk drive	(DL) = Drive number (0 = A, 1 = B, 2 = C)	(AL) = Number of disk drives (2 for single drive)
19	Get default drive code	None	(AL) = Default drive code (0 = A, 1 = B, 2 = C)
2E	Set verify state	(DL) = 0 (AL) = 0 to turn verify off = 1 to turn verify on	None
Date and Time Functions			
2A	Get date	None	(CX) = Year (1980-2099) (DH) = Month (1-12) (DL) = Day (1-31)
2B	Set date	(CX), (DX) = Date, in same format as function 2A	(AL) = 0 if date is valid = FF if date is invalid
2C	Get time	None	(CH) = Hours (0-23) (CL) = Minutes (0-59)

Note: See the DOS 1.1 manual of the *DOS Technical Reference* manual for other disk functions in the range (AH) = F through (AH) = 2F. DOS 2 users should use the Extended File Management functions.

Table 6-5. Function calls with the Type 21 interrupt (continued).

(AH) Operation	*Additional Input Registers*	*Result Registers*
		(DH) = Seconds (0-59) (DL) = 1/100 Seconds (0-99)
2D Set time	(CX), (DX) = Time, in same format as function 2C	(AL) = 0 if time is valid = FF if time is invalid
Interrupt Vector Functions		
25 Set interrupt vector	(DS:DX) = Vector address (AL) = Interrupt number (type)	None
35 Read interrupt vector address	(AL) = Interrupt number (type)	(ES:BX) = Vector address
Directory Functions (DOS 2 only)		
	Note: For "error returned" codes, see Table 6-6.	
39 Create a directory (MKDIR)	(DS:DX) = Address of ASCIIZ string for directory	Error returned is 3 or 5.
3A Remove a directory (RMDIR)	(DS:DX) = Address of ASCIIZ string for directory	Error returned is 3 or 5.
3B Change the directory (CHDIR)	(DS:DX) = Address of ASCIIZ string for new directory	Error returned is 3.
47 Get current directory	(DL) = Drive number (0 = default, 1 = A, etc.) (DS:SI) = Address of 64-byte buffer	(DS:SI) = Address of ASCIIZ string Error returned is 15.
Extended File Management Functions (DOS 2 only)		
Note: For "error returned" codes, see Table 6-6.		
36 Get free disk space	(DL) = Drive number (0 = default, 1 = A, etc.)	(AX) = 0FFFFH if invalid = Sectors per cluster (BX) = No. of free clusters (DX) = Total no. of clusters (CX) = Bytes per sector
3C Create a file	(DS:DX) = Address of ASCIIZ string (CX) = Attribute of file	(AX) = File handle Error returned is 3, 4, or 5.

Table 6-5. Function calls with the Type 21 interrupt (continued).

(AH)	*Operation*	*Additional Input Registers*	*Result Registers*
3D	Open a file	(DS:DX) = Address of ASCIIZ string (AL) = 0 to open for reading = 1 to open for writing = 2 to open for reading and writing	(AX) = File handle Error returned is 2, 4, 5, or 12
3E	Close a file handle	(BX) = File handle	Error returned is 6.
3F	Read from file or device	(BX) = File handle (CX) = No. of bytes to read (DS:DX) = Buffer address	(AX) = No. of bytes read = 0 if read from end of file Error returned is 5 or 6.
40	Write to a file or device	(BX) = File handle (CX) = No. of bytes to write (DS:DX) = Buffer address	(AX) = No. of bytes written Error returned is 5 or 6.
41	Delete a file	(DS:DX) = Address of ASCIIZ string	Errors returned are 2 or 5.
43	Get file attribute	(AL) = 0 (DS:DX) = Address of ASCIIZ string for file	(CX) = Attribute Error returned is 2 or 5.
43	Set file attribute	(AL) = 1 (DS:DX) = Address of ASCIIZ string for file (CX) = Attribute	Error returned is 2 or 5.
54	Get verify state	None	(AL) = 0 if verify is off = 1 if verify is on
56	Rename a file	(DS:DX) = Address of ASCIIZ string for old name (ES:DI) = Address of ASCIIZ string for new name	Error returned is 3, 5, or 17.

new function calls (see *Directory Functions* and *Extended File Management Functions)* return CF = 0 if the operation is successful and CF = 1 if an error occurred. Along with CF = 1, an error code is returned in AX; Table 6-6 tells what these error codes mean.

Type 21 Function Calls

Keyboard Functions

The keyboard functions are straightforward; they read individual key characters into AL or a sequence of characters (a *string*) into memory. Unless you're looking for some unusual key combinations (more about this in Section 6.4), you should find these functions convenient to use.

Table 6-6. Error codes for DOS 2 function calls.

Code	*Meaning*
1	Invalid function number
2	File not found
3	Path not found
4	Too many open files (no handles left)
5	Access denied
6	Invalid handle
7	Memory control blocks destroyed
8	Insufficient memory
9	Invalid memory block address
10	Invalid environment
11	Invalid format
12	Invalid access code
13	Invalid data
15	Invalid drive was specified
16	Attempted to remove the current directory
17	Not same device
18	No more files

Interactive programs often require the user to respond to a prompt or select from a menu by typing a single letter or number. For example, suppose your program displays a message that tells the user to press either *Y* or *N* (perhaps to continue or stop). Y makes the program jump to a set of instructions labeled YES; N makes it jump to instructions labeled NO. Any other key makes the program wait for either Y or N. This sequence does the job:

```
GET_KEY:  MOV   AH,1          ;Read a key
          INT   21H
          CMP   AL,'Y'        ;Is it Y?
          JE    YES           ; If so, jump to YES
          CMP   AL,'N'        ;Is it N?
          JE    NO            ; If so, jump to NO
          JNE   GET_KEY       ; Otherwise, wait for Y or N
```

This particular sequence accepts only an uppercase Y or N, but users are inclined to press a key without pressing Shift. To make the program accept lowercase responses, add the additional test instructions *CMP AL,'y'* and *CMP AL,'n'*.

Many applications require the user to enter a string, such as a name or address. Function A does this for you. To use function A, you must reserve space in the data segment for the string. Specifically, you need a block of bytes equal to the maximum number of characters plus two, with the character count in the first byte. For example, to reserve space for a string of up to 50 characters, you would enter this statement in the data segment:

```
USER_STRING   DB   50,51 DUP(?)
```

The string-reading instructions are:

```
LEA   DX,USER_STRING   ;Make DX point to buffer
MOV   AH,0AH           ;Read the string
INT   21H
```

Function A puts the count of the characters received (not including Return) in the second byte of the string and leaves DS:DX pointing to the maximum count byte. In many cases, you want the character count in the CX register and want DS:DX to point to the first string character. READ_KEYS, the procedure in Example 6-2, does that job.

Display Functions

There are also functions that display or print a character in AL and one that displays a string. The string display function, 9, requires the string to have a dollar sign at the end. Hence, to produce the prompt *Please enter your name:* , enter the following statement in the data segment:

```
PROMPT   DB   'Please enter your name: $'
```

and use these instructions in your program:

```
LEA   DX,PROMPT
MOV   AH,9
INT   21H
```

Example 6-2. Read a string from the keyboard.

```
; This procedure reads up to 50 keystrokes.  It returns the
; string address in DS:DX and the character count in CX.
; No other registers are affected.
;
; Put this statement in the data segment:
;
USER_STRING   DB   50,51 DUP(?)
;
; Here is the actual procedure.
;
READ_KEYS   PROC
      PUSH   AX
      LEA    DX,USER_STRING      ;Read the string
      MOV    AH,0AH
      INT    21H
      SUB    CH,CH               ;Put character count in CX
      MOV    CL,USER_STRING+1
      ADD    DX,2                ;Make DX point to string
      POP    AX
      RET
READ_KEYS   ENDP
```

This leaves the cursor one space past the colon, which is where you want the user's name to appear when he or she types it.

To make the cursor advance to the next line after displaying a string (e.g., the string is a message rather than a prompt), precede the $ with Carriage Return and Line Feed characters, as in

```
MESSAGE   DB   'The sort operation is finished.',0DH,0AH,'$'
```

We discuss these and other control characters in Section 6.4.

File Management Functions

The *File Management Functions* listed in Table 6-5 are a small portion of a large variety of functions that can be used with any version of DOS. Besides the functions we show, there are others that open and close files, delete files,

rename files, etc. However, these operations involve File Control Blocks (FCBs), and can get quite complicated. With the advent of DOS 2.0, IBM provides us with much easier functions that do the same job—the so-called *Extended File Management Functions.* We'll discuss them shortly.

Date and Time Functions

You probably won't want to change the date from within an assembly language program, but having access to the time can be quite useful. For example, to find out how long a procedure takes to execute, you can set the time to zero, CALL the procedure, then read the time value; this result is the elapsed time. The following sequence calculates the execution time of a procedure called SORT:

```
SUB     CX,CX       ;Set time to zero
SUB     DX,DX
MOV     AH,2DH
INT     21H
CALL    SORT        ;Execute the procedure
MOV     AH,2CH      ;Read the elapsed time
INT     21H
```

To do the same thing without disturbing the computer's time setting, you can subtract the start time from the end time, using temporary data segment locations to save the start time, as follows:

```
        MOV     AH,2CH      ;Read the start time
        INT     21H
        MOV     HRS,CH      ; and save it
        MOV     MINS,CL
        MOV     SECS,DH
        MOV     HSECS,DL
        CALL    SORT        ;Execute the procedure
        MOV     AH,2CH      ;Read the end time
        INT     21H
        SUB     DL,HSECS    ;Calculate the difference
        JNC     SUB_SECS
        ADD     DL,100
        DEC     DH
SUB_SECS:
        SUB     DH,SECS
        JNC     SUB_MINS
        ADD     DH,60
        DEC     CL
SUB_MINS:
        SUB     CL,MINS
```

```
                JNC     SUB_HRS
                ADD     CL,60
                DEC     CH
SUB_HRS:        SUB     CH,HRS
                JNC     DONE
                ADD     CH,24
DONE:           RET
```

The extra instructions after each subtraction convert the result to a positive number.

The time functions are especially useful for generating delays. You will want to generate a delay to make the computer produce sounds through its speaker, or when you display graphics shapes on the screen. For sound, the delay determines the duration of the note; for graphics, it determines how long a shape appears or how long the computer waits before displaying the next shape.

Programs that generate delays usually do the following:

1. Read the current time.
2. Add the current time to the specified time increments, to produce a "target" time.
3. Adjust the target time so that hours do not exceed 24, minutes and seconds do not exceed 60, and hundredths of a second do not exceed 100. This involves carrying any excess to the next higher unit.
4. Read the time repeatedly until the current time matches or exceeds the target time.

Figure 6-4 shows a flowchart to produce a time delay based on input values for minutes, seconds, and hundredths of a second. (We assume you won't want to delay for hours at a time, although you could.) Example 6-3 shows a general-purpose delay procedure (DELAY) that uses this approach.

Example 6-3. Generate a specified delay.

```
;  This procedure makes the processor wait for a specified
;  interval.  Enter with minutes in AL, seconds in BH, and
;  hundredths in BL.
;  No registers are affected.
;
DELAY   PROC
        PUSH    AX              ;Save affected registers
        PUSH    BX
        PUSH    CX
        PUSH    DX
```

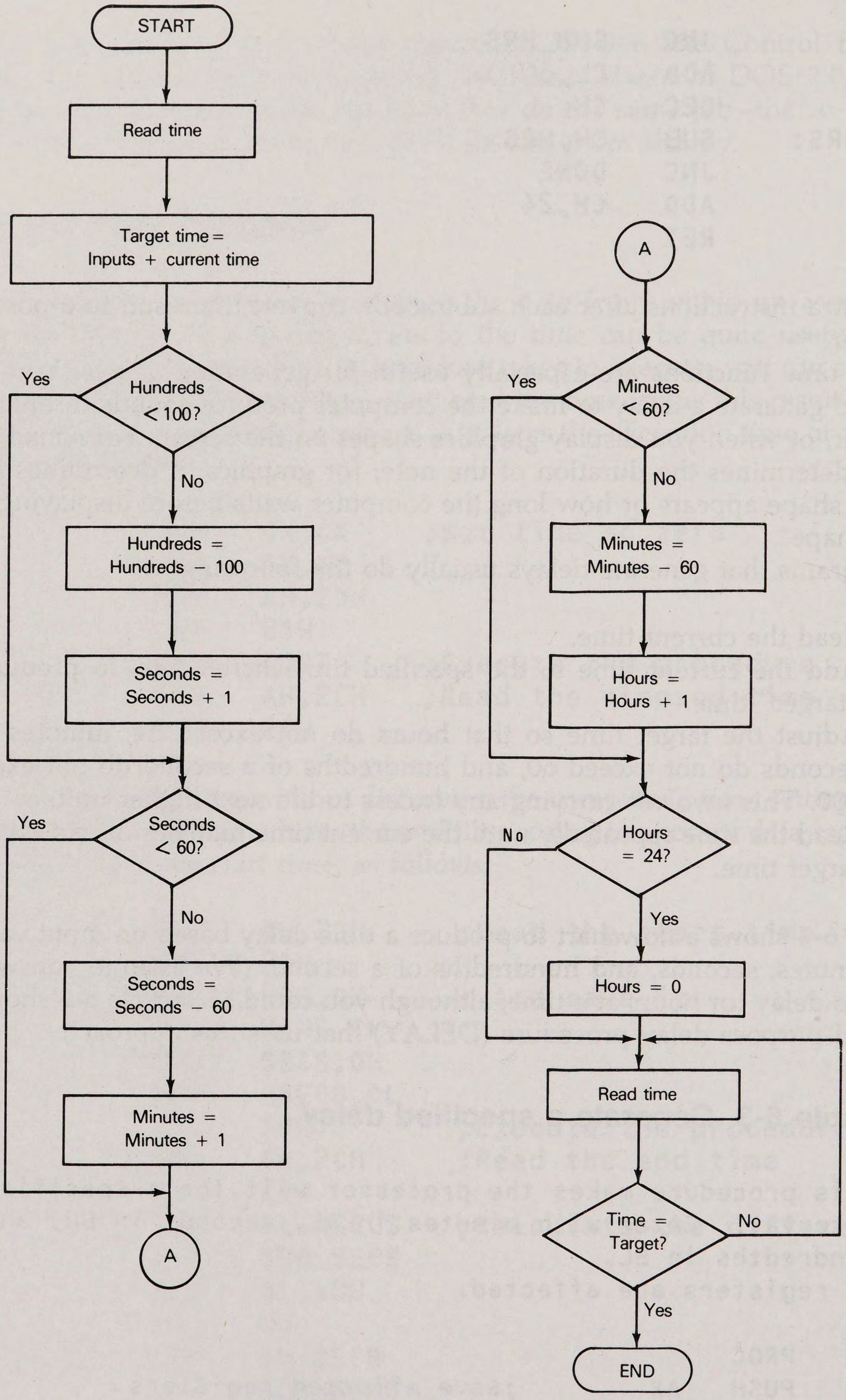


Figure 6-4. Delay flowchart.

```
        MOV     AH,2CH      ;Read the current time
        INT     21H
;
; Add the current time to the input values.
;
        MOV     AH,CH       ;Hours
        ADD     AL,CL       ;Minutes
        ADD     BH,DH       ;Seconds
        ADD     BL,DL       ;Hundredths
;
; Adjust the results to produce a legal target time.
;
        CMP     BL,100      ;Hundredths must be < 100
        JB      SECS
        SUB     BL,100
        INC     BH
SECS:   CMP     BH,60       ;Minutes must be < 60
        JB      MINS
        SUB     BH,60
        INC     AL
MINS:   CMP     AL,60       ;Seconds must be < 60
        JB      HRS
        SUB     AL,60
        INC     AH
HRS:    CMP     AH,24       ;Hours must be < 24
        JNE     CHECK
        SUB     AH,AH
;
; Wait until time matches target.
;
CHECK:  PUSH    AX          ;Read the time again
        MOV     AH,2CH
        INT     21H
        POP     AX
        CMP     CX,AX       ;Compare hours and minutes
        JA      QUIT
        JB      CHECK
        CMP     DX,BX       ;Compare seconds and hundredths
        JB      CHECK
QUIT:   POP     DX          ;Restore registers
        POP     CX
        POP     BX
        POP     AX
        RET
DELAY   ENDP
```

Interrupt Vector Functions

Functions 25 and 35 let you operate on the interrupt vector addresses. For example, to add an interrupt service routine called NEW_INT to the system and have it respond to *INT 60H*, use this sequence:

```
MOV   AL,60H                  ;Put interrupt type in AL
MOV   DX,SEG NEW_INT          ;Put segment number in DS
MOV   DS,DX
MOV   DX,OFFSET NEW_INT       ; and offset in DX
MOV   AH,25H                  ;Select function call 25
INT   21H                     ;Change the interrupt vector
```

Remember, an interrupt service routine is simply a procedure with an IRET (rather than a RET) at the end. Hence, NEW_INT has the form

```
NEW_INT   PROC   FAR
      ..    (Instructions in the interrupt service procedure)
      ..
      IRET
NEW_INT   ENDP
```

Directory Functions

These functions let you read the directory name and perform the DOS 2 *MKDIR, RMDIR,* and *CHDIR* operations from within a program. Each reports on the outcome by setting CF to either 0 (success) or 1 (error). Further, if CF is 1, then AX contains an error code.

In each case, you must provide an ASCII text string that describes the directory. It contains an optional drive specifier (e.g., *C:*), the path, and a byte containing the value zero (not the ASCII code for 0). Because of the zero byte, IBM calls this an "ASCIIZ" string. For example, to switch to a subdirectory called *SALES* off the root directory, put the following statement in the data segment:

```
SALES_DIR   DB   '\SALES',0
```

The code segment would contain

```
        LEA   DX,SALES_DIR    ;Point to subdirectory name
        MOV   AH,3BH          ;Change the subdirectory
        INT   21H
        JNC   OKAY            ;Successful?
        ..                    ;No.  Read error code from AX
        ..
OKAY:
```

Extended File Management Functions

Like the Directory Functions, most Extended File Management Functions require an ASCIIZ string—in this case, one that identifies the file. The ASCIIZ string for a file contains the drive name and path (if required), followed by the filename, extension (if any), and a zero byte. For example, the data segment statement for the file SALESQ4.NE in the SALES directory may have this form:

```
SALESQ4$  DB    '\SALES\SALESQ4.NE',0
```

The Extended File Management Functions involve two terms we haven't encountered: *attribute* and *handle.* An attribute is DOS 2's way to classify entries on a disk. Each entry has an *attribute byte* whose bits tell whether the entry is a read-only file (bit 0 = 1), a hidden file (bit 1 = 1), a system file (bit 2 = 1), a volume label (bit 3 = 1), a subdirectory name (bit 4 = 1), or an ordinary file (bit 5). Bit 5 is the "archive" bit for fixed disk files; DOS sets it to 1 when you change the file and clears it to 0 when you use the BACKUP command to copy the file to a floppy disk. For floppy disk files, bit 5 is always 1.

Function 43 lets you read or change a file's attribute. Generally you won't want to change the attribute of a volume label or a subdirectory, but you could, say, make a system file hidden. (By hidden, we mean that the file doesn't appear in normal DIR listings.) However, the most important thing you can do with function 43 is make a file *read-only.* Read-only files are "write-protected"; they cannot be changed or deleted.

With DOS 2, using Function 43 is the only way you can write-protect a file! (DOS 3's ATTRIB command can write-protect files, but DOS 2 has no such command.) We will provide programs that write-protect files and remove protection from them under "Write-Protecting Files" later in this section.

A handle is a number that identifies an open file or an I/O device. It is similar to the number you assign with a BASIC statement such as *OPEN "DATA" FOR OUTPUT AS #1.* When you start DOS 2, it assigns the following handles:

- *0* is the standard input device. DOS assigns handle 0 to the keyboard, but you can reassign or "redirect" it to some other device (see "Redirection of Standard Input and Output Devices" in Chapter 1 of the *DOS* manual). You may, for example, want to redirect 0 to a remote terminal.
- *1* is the standard output device. DOS assigns handle 1 to the display screen, but you can redirect it (see the preceding reference).
- *2* is the standard error output device. It cannot be redirected.
- *3* is the standard auxiliary device.
- *4* is the standard printer.

DOS is set up to accept four additional, user-defined handles (for a total of eight), but you may increase this number.

For example, you can use function 40 to display a message on the screen, the standard output device. Your data segment would contain the message itself, as follows:

```
MSG   DB   "Please try again."  ;Message
      DB   13,10                ;Carriage return and line feed
MSGL  EQU  $-MSG                ;Length of message
```

Your code segment would contain these message-writing instructions:

```
LEA   DX,MSG     ;Put message offset in DX
MOV   CX,MSGL    ;CX counts characters to display
MOV   BX,1       ;Use file handle for screen
INT   21H        ;Display the message
```

Note that the message string does not have to end with a $ character, as it does for function 9 (see *Keyboard Functions*); we use MSGL EQU $-MSG to calculate the length.

DOS 2 Error Message Program

As we mentioned earlier, most of the new DOS 2 function calls report errors by setting the Carry Flag (CF) to 1 and loading an error code into AX. We summarized these errors in Table 6-6. To save you from having to look up the error each time, Example 6-4 shows a procedure called SHOW_ERR that uses an error code in AX to display a message telling you which error occurred.

If your calling program has a data segment with a different name than SHOW_ERR's, be sure to save the calling program's DS value. That is, insert the following kinds of instructions at the beginning of SHOW_ERR's code segment:

```
PUSH  DS         ;Save caller's DS value
MOV   SI,DSEG    ;Initialize DS
MOV   DS,SI
```

Of course, you must also insert a POP DS instruction at the end of the code segment.

Example 6-4. DOS 2 error message program.

```
; Displays a message based on an error code in AX.
; No registers are affected.

Put these values in the data segment:
CR    EQU  13
LF    EQU  10
EOM   EQU  '$'
OUT_OF_RANGE  DB  'Error code is not in valid range (1-18)'
              DB  CR,LF,EOM
ER1   DB    'Invalid function number',CR,LF,EOM
ER2   DB    'File not found',CR,LF,EOM
ER3   DB    'Path not found',CR,LF,EOM
ER4   DB    'Too many open files (No handles left)',CR,LF,EOM
ER5   DB    'Access denied',CR,LF,EOM
ER6   DB    'Invalid handle',CR,LF,EOM
ER7   DB    'Memory control blocks destroyed',CR,LF,EOM
ER8   DB    'Insufficient memory',CR,LF,EOM
ER9   DB    'Invalid memory block address',CR,LF,EOM
ER10  DB    'Invalid environment',CR,LF,EOM
ER11  DB    'Invalid format',CR,LF,EOM
ER12  DB    'Invalid access code',CR,LF,EOM
ER13  DB    'Invalid data',CR,LF,EOM
ER14  DB    'No such message',CR,LF,EOM
ER15  DB    'Invalid drive was specified',CR,LF,EOM
ER16  DB    'Attempted to remove current directory',CR,LF,EOM
ER17  DB    'Not same device',CR,LF,EOM
ER18  DB    'No more files',CR,LF,EOM
ERTAB DW  ER1,ER2,ER3,ER4,ER5,ER6,ER7,ER8,ER9
      DW  ER10,ER11,ER12,ER13,ER14,ER15,ER16,ER17,ER18

Here is the main procedure:
SHOW_ERR  PROC
          PUSH  AX              ;Save input error code
          PUSH  BX              ;Save other working registers
          PUSH  CX
          CMP   AX,18           ;Make sure error code is < 18
          JG    O_O_R
          CMP   AX,0
          JG    IN_RANGE
O_O_R:    LEA   DX,OUT_OF_RANGE
          JMP   SHORT DISP_MSG
```

```
IN_RANGE: LEA   BX,ERTAB-2      ;Point to table of offsets
          SHL   AX,1            ;Point to correct offset
          ADD   BX,AX
          MOV   DX,[BX]         ;Put message address into DX
DISP_MSG: MOV   AH,9            ;Display message string
          INT   21H
          POP   DX
          POP   BX
          POP   AX
          RET
SHOW_ERR  ENDP
```

Write-Protecting Files

As we mentioned earlier in this section, the DOS 2 function 43 lets you change the attribute of a file. Among other things, it allows you to write-protect files. Once a file has been write-protected, users can read its contents, but they can neither write to it nor erase it. Attempting to erase a write-protected or *read-only* file makes DOS produce the message "File not found."

Example 6-5 shows a program called LOCK that write-protects a specified file. LOCK works like DOS in that you must enter a drive name and path definition if the file is on a different drive or directory. Note that LOCK calls SHOW_ERR to report any errors. Example 6-6 shows a nearly identical program called UNLOCK, which removes write-protection.

Example 6-5. Write-protect a file.

```
; This procedure write-protects a file by setting the
; read-only bit in its attribute byte.

     EXTRN      SHOW_ERR:FAR

; Put the following in the data segment:

PROMPT     DB   'Which file do you want to lock? '
PROMPTL    DB   $-PROMPT
FILESPEC   DB   60 DUP(?)

; Here is the main procedure:

LOCK  PROC
      MOV   AH,40H              ;Display prompt
```

```
        MOV   BX,1
        MOV   CX,PROMPTL
        LEA   DX,PROMPT
        INT   21H
        MOV   AH,3FH              ;Obtain filespec from user
        MOV   BX,0
        MOV   CX,60
        LEA   DX,FILESPEC
        INT   21H

;Convert entry to ASCIIZ string by replacing CR with 0.

        SUB   AX,2                ;Point to CR character
        MOV   BX,AX               ;Put pointer in BX
        MOV   FILESPEC[BX]        ;Replace CR with 0

;Change the file to read-only.

        MOV   AH,43H              ;Read the attribute byte
        MOV   AL,0
        LEA   DX,FILESPEC
        INT   21H
        JNC   SET_ATT             ;Read error?
        CALL  SHOW_ERR            ; If so, display error message
        RET
SET_ATT: OR   CX,1                ; Otherwise, set the read-only bit
        MOV   AL,1                ;Write the attribute byte
        MOV   AH,43H
        LEA   DX,FILESPEC
        INT   21H
        JNC   LEAVE               ;Write error?
        CALL  SHOW_ERR            ; If so, display error message
LEAVE: RET
LOCK   ENDP
```

Example 6-6. Remove write-protection from a file.

```
; This procedure removes write-protection by clearing the
; read-only bit in a file's attribute byte.

        EXTRN     SHOW_ERR:FAR

; Put the following in the data segment:
```

```
PROMPT    DB  'Which file do you want to unlock? '
PROMPTL   DB  $-PROMPT
FILESPEC  DB  60 DUP(?)

; Here is the main procedure:

UNLOCK  PROC
      MOV   AH,40H            ;Display prompt
      MOV   BX,1
      MOV   CX,PROMPTL
      LEA   DX,PROMPT
      INT   21H
      MOV   AH,3FH            ;Obtain filespec from user
      MOV   BX,0
      MOV   CX,60
      LEA   DX,FILESPEC
      INT   21H

;Convert entry to ASCIIZ string by replacing CR with 0.

      SUB   AX,2              ;Point to CR character
      MOV   BX,AX             ;Put pointer in BX
      MOV   FILESPEC[BX]      ;Replace CR with 0

;Change the file from read-only to read/write.

      MOV   AH,43H            ;Read the attribute byte
      MOV   AL,0
      LEA   DX,FILESPEC
      INT   21H
      JNC   CLR_ATT           ;Read error?
      CALL  SHOW_ERR          ; If so, display error message
      RET
CLR_ATT: AND  CX,0FEH         ; Otherwise, clear the read-only bit
      MOV   AL,1              ;Write the attribute byte
      MOV   AH,43H
      LEA   DX,FILESPEC
      INT   21H
      JNC   LEAVE             ;Write error?
      CALL  SHOW_ERR          ; If so, display error message
LEAVE: RET
UNLOCK  ENDP
```

6.4 Keyboard Operations

This section gives a brief (and simplified) explanation of how the keyboard works and suggests ways you may communicate with it.

ASCII

Many microcomputer keyboards connect to the computer through a chip called an "encoder" that translates each keystroke into an 8-bit *ASCII* code. ASCII (pronounced "ass-key") is a set of numeric codes that computers use to transmit data. As Table 6-7 shows, there are 128 codes in the ASCII character set, ranging from 00H to 7FH. To find the ASCII code for a character, you combine the most-significant digit (MSD) from the top row with the least-significant digit (LSD) from the left-hand column. For example, the letter "A" has an MSD of 4 and an LSD of 1, so its code is 41H.

Note that besides the usual letters, numbers, and symbols on the keyboard, the ASCII character set includes control characters such as CR (Car-

Table 6-7. Standard ASCII character set.

LSD \ MSD		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	•	>	N	↑	n	~
F	1111	SI	US	/	?	O	←	o	DEL

NUL — Null
SOH — Start of Heading
STX — Start of Text
ETX — End of Text
EOT — End of Transmission
ENQ — Enquiry
ACK — Acknowledge
BEL — Bell
BS — Backspace
HT — Horizontal Tabulation
LF — Line Feed
VT — Vertical Tabulation
FF — Form Feed
CR — Carriage Return
SO — Shift Out
SI — Shift In

DLE — Data Link Escape
DC — Device Control
NAK — Negative Acknowledge
SYN — Synchronous Idle
ETB — End of Transmission Block
CAN — Cancel
EM — End of Medium
SUB — Substitute
ESC — Escape
FS — File Separator
GS — Group Separator
RS — Record Separator
US — Unit Separator
SP — Space (Blank)
DEL — Delete

riage Return), FF (Form Feed), and LF (Line Feed). It also includes some characters that are used in communications, such as ACK (Acknowledge), STX (Start of Text), and EOT (End of Transmission). However, if you study Table 6-7 for any amount of time (granted, it doesn't make very interesting reading) you will note that it does not include the PC's special keys. For example, you won't find codes for the function keys or the numeric keypad. This presented a special problem for the keyboard designers at IBM, and they came up with a rather interesting solution that's worth discussing.

How the PC Keyboard Works

Here is how the PC keyboard works. Whenever you press or release a key, the keyboard unit saves a byte in its internal memory. This byte contains a "pressed" or "released" code (1 or 0, respectively) in Bit 7 and a key identifier or *scan code* in the remaining bits. There are 83 possible scan codes, one for each key on the keyboard (see Figure 6-5). Once the keyboard unit has stored something in its memory, it sends the PC's BIOS a Type 9 interrupt, saying "I have something for you."

The keyboard's memory can hold up to 20 bytes, or 10 press-and-release operations (i.e., keystrokes). This allows you to type even while the computer's microprocessor is busy, as it is while dumping the screen to the printer, for example.

If interrupts are enabled, BIOS' Type 9 interrupt routine (KB_INT) reads bytes from the keyboard unit's memory and translates them into character codes. As part of the translation process, KB_INT checks to see if there have been several "presses" without a "release"; this lets it produce character codes that reflect Shift, Ctrl, or Alt being held down while you press a second key. Upon completion, the Type 9 routine stores the scan code and the character code in a "keyboard buffer" in memory.

With space for 15 keystrokes, the buffer is big enough, and the software fast enough, to keep up with the fastest typists. But if you somehow manage to press a key when the buffer is full, BIOS ignores the key and beeps the speaker.

Extended ASCII

As we mentioned earlier, because the PC has so many more keys and so many keys that differ from a standard computer, the character codes cannot be represented by the standard set of 128 ASCII codes. In all, the PC provides for 256 codes, which IBM calls Extended ASCII.

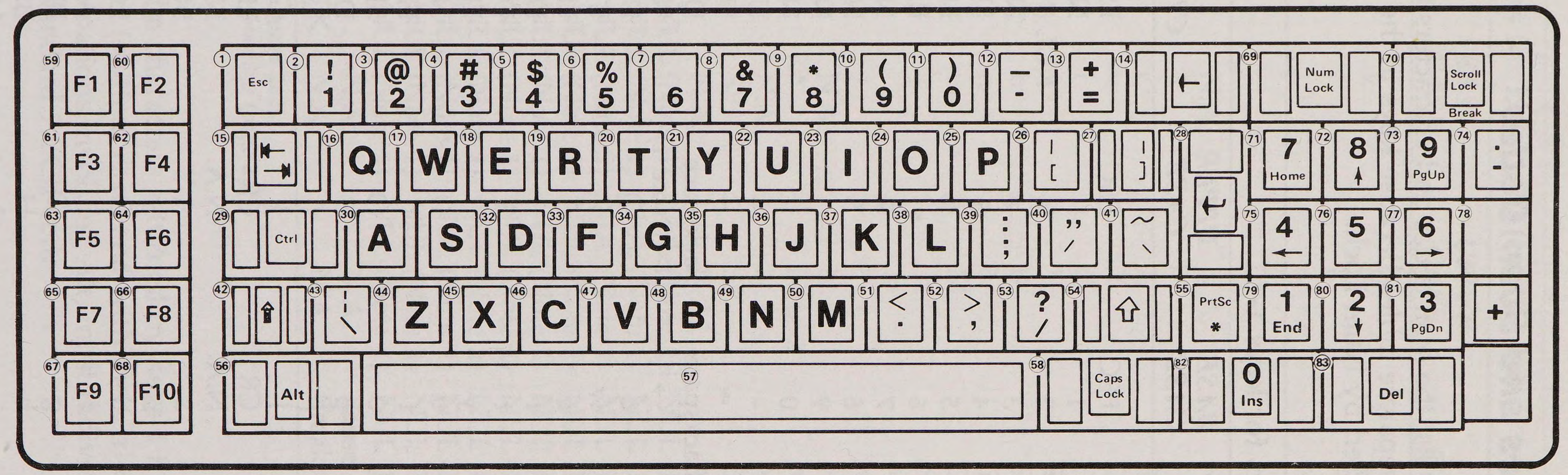


Figure 6-5. The keyboard and its scan codes.

Character Codes and Scan Codes

Tables 6-8 and 6-9 list the scan codes and characters that the 83 keys produce. We need two separate tables because keys in the numeric keypad (71 through 83) are affected by Num Lock.

Table 6-8. Characters for keys 1-70.

SCAN CODE		*BASE*	*UPPER*		
Dec	*Hex*	*CASE*	*CASE*	*CTRL*	*ALT*
1	1	ESC	ESC	ESC	N.A.
2	2	1	!	N.A.	Note 1
3	3	2	@	NUL (Note 1)	Note 1
4	4	3	#	N.A.	Note 1
5	5	4	$	N.A.	Note 1
6	6	5	%	N.A.	Note 1
7	7	6	^	RS	Note 1
8	8	7	&	N.A.	Note 1
9	9	8	*	N.A.	Note 1
10	A	9	(	N.A.	Note 1
11	B	0	)	N.A.	Note 1
12	C	-	_	US	Note 1
13	D	=	+	N.A.	Note 1
14	E	Backspace	Backspace	DEL	N.A.
15	F	→|	|← (Note 1)	N.A.	N.A.
16	10	q	Q	DC1	Note 1
17	11	w	W	ETB	Note 1
18	12	e	E	ENQ	Note 1
19	13	r	R	DC2	Note 1
20	14	t	T	DC4	Note 1
21	15	y	Y	EM	Note 1
22	16	u	U	NAK	Note 1
23	17	i	I	HT	Note 1
24	18	0	O	SI	Note 1
25	19	p	P	DLE	Note 1
26	1A	[	{	ESC	N.A.
27	1B	]	}	GS	N.A.
28	1C	CR	CR	LF	N.A.
29	1D CTRL	N.A.	N.A.	N.A.	N.A.
30	1E	a	A	SOH	Note 1
31	1F	s	S	DC3	Note 1
32	20	d	D	EOT	Note 1
33	21	f	F	ACK	Note 1
34	22	g	G	BEL	Note 1
35	23	h	H	BS	Note 1
36	24	j	J	LF	Note 1

Table 6-8. Characters for keys 1-70 (continued).

SCAN CODE		*BASE*	*UPPER*		
Dec	*Hex*	*CASE*	*CASE*	*CTRL*	*ALT*
37	25	k	K	VT	Note 1
38	26	l	L	FF	Note 1
39	27	;	:	N.A.	N.A.
40	28	'	"	N.A.	N.A.
41	29	`	~	N.A.	N.A.
42	2A SHIFT	N.A.	N.A.	N.A.	N.A.
43	2B	\	|	FS	N.A.
44	2C	z	Z	SUB	Note 1
45	2D	x	X	CAN	Note 1
46	2E	c	C	ETX	Note 1
47	2F	v	V	SYN	Note 1
48	30	b	B	STX	Note 1
49	31	n	N	SO	Note 1
50	32	m	M	CR	Note 1
51	33	,	<	N.A.	N.A.
52	34	.	>	N.A.	N.A.
53	35	/	?	N.A.	N.A.
54	36 SHIFT	N.A.	N.A.	N.A.	N.A.
55	37	*	Note 2	Note 1	N.A.
56	38 ALT	N.A.	N.A.	N.A.	N.A.
57	39 SPACE BAR	SP	SP	SP	SP
58	3A CAPS LOCK	N.A.	N.A.	N.A.	N.A.
59	3B F1	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
60	3C F2	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
61	3D F3	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
62	3E F4	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
63	3F F5	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
64	40 F6	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
65	41 F7	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
66	42 F8	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
67	43 F9	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
68	44 F10	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)	NUL (Note 1)
69	45 NUM LOCK	N.A.	N.A.	Pause (Note 2)	N.A.
70	46 SCROLL LOCK	N.A.	N.A.	Break (Note 2)	N.A.

Notes: 1. See *Extended Codes* (page 242).
2. See *Special Key Combinations* (page 243).

Table 6-8 shows the characters returned by keys 1 through 70 in the base (unshifted) and upper (shifted) case, and with the Ctrl or Alt key depressed. Except for key combinations involving Alt, most individual keys and combinations generate a standard ASCII code.

Table 6-9. Characters for keys 71-83 (the numeric keypad).

SCAN CODE					
Dec	*Hex*	*NUM LOCK*	*BASE CASE*	*ALT*	*CTRL*
71	47	7	Home (Note 1)	N.A.	Clear Screen
72	48	8	↑ (Note 1)	N.A.	N.A.
73	49	9	Page Up (Note 1)	N.A.	Top of Text and Home
74	4A	-	—	N.A.	N.A.
75	4B	4	← (Note 1)	N.A.	Reverse Word (Note 1)
76	4C	5	N.A.	N.A.	N.A.
77	4D	6	→ (Note 1)	N.A.	Adv Word (Note 1)
78	4E	+	+	N.A.	N.A.
79	4F	1	End (Note 1)	N.A.	Erase to EOL (Note 1)
80	50	2	↓ (Note 1)	N.A.	N.A.
81	51	3	Page Down (Note 1)	N.A.	Erase to EOS (Note 1)
82	52	0	INS	N.A.	N.A.
83	53		DEL (Notes 1,2)	Note 2	Note 2

Notes: 1. See *Extended Codes* (page 242).
2. See *Special Key Combinations* (page 243).

Keys that have no ASCII equivalent (such as Alt and the function keys) return 0 for the character code. The 0 indicates that your program must examine a second code to find out which key or keys was pressed. We will discuss these so-called *Extended Codes* shortly. Other non-ASCII keys produce special operations, such as printing the screen contents. We summarize them under *Special Key Combinations.* Finally, there are some key combinations that BIOS simply ignores; Table 6-9 shows them as *N.A.*, for Not Available.

Extended Codes

When you read a key in from the keyboard, BIOS returns its character code and scan code in AL and AH, respectively. Non-ASCII keys return 0 in AL and an "extended code" in AH. Table 6-10 lists the extended codes and shows which key combinations they represent. Note that extended codes between 3 and 53 correspond to the scan code.

Table 6-10. Keyboard extended codes.

EXTENDED CODE	*FUNCTION*
3	CTRL @
F	←
10-19	ALT Q, W, E, R, T, Y, U, I, O, P (second row)
1E-26	ALT A, S, D, F, G, H, J, K, L (third row)
2C-32	ALT Z, X, C, V, B, N, M (fourth row)
3B-44	F1-F10 function keys (base case)
47	Home
48	↑
49	Page Up and Home Cursor
4B	←
4D	→
4F	END
50	↓
51	Page Down and Home Cursor
52	INS
53	DEL
54-5D	SHIFT F1-F10
5E-67	CTRL F1-F10
68-71	ALT F1-F10
72	CTRL PRTSC (start/stop echo to printer)
73	CTRL ← (reverse word)
74	CTRL → (advance word)
75	CTRL END (erase to end of line)
76	CTRL PG DN (erase to end of screen)
77	CTRL HOME (clear screen and home)
78-83	ALT 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, = (top row)
84	CTRL PG UP (Top 25 lines of text and home cursor)

Special Key Combinations

The following combinations of keys produce special effects:

- *Alt-Ctrl-Del* restarts or "reboots" the computer.
- *Ctrl-Break* invokes the Type 1B (Keyboard Break) interrupt. This key combination returns AL = 0 and AH = 0.
- *Ctrl-Num Lock* makes the computer wait for you to press any key but Num Lock. This gives you a way to suspend an operation (e.g., listing a program), then resume.
- *Shift-PrtSc* invokes the Type 5 (Print Screen) interrupt.
- The BIOS keyboard routine treats Ctrl, Shift, Num Lock, Scroll Lock, and Ins as control keys. The service routine for the type 16 (Keyboard

I/O) interrupt returns a "shift status" byte that tells you when one of these keys is pressed.

Keyboard Interrupts

Earlier in this chapter, we mentioned two different kinds of interrupts you can use to communicate with the keyboard: the Type 16 (Keyboard I/O) interrupt that is built into the BIOS and the keyboard functions that DOS contributes within its Type 21 function calls. Let's look at each to find out which to use.

Type 16 Interrupt Operations

The Type 16 (Keyboard I/O) interrupt provides very primitive keyboard operations that can be useful if you are designing a program that runs independent of DOS. Type 16 lets you select from three different operations, based on a value in AH:

- If AH = 0, Type 16's interrupt service routine, KEYBOARD_IO, reads the scan code of the next key in the keyboard buffer into AH and its character code into AL, then advances the buffer pointer. If the buffer is empty, KEYBOARD_IO waits for a key to be pressed before proceeding.
- If AH = 1, KEYBOARD_IO returns the status of the keyboard buffer in the Zero Flag (ZF). If the buffer is empty, ZF is 1. If any key codes are available for reading, ZF is 0, which indicates that the next available character is in AX and the entry remains in the buffer.
- If AH = 2, KEYBOARD_IO returns a keyboard status byte in AL. A description of this byte follows.

The KEYBOARD_IO routine affects only AX and the flags.

The upper half of Figure 6-6 shows the arrangement of the status byte returned by the AH = 2 option. The upper four bits of this byte (KB_FLAG in the BIOS listing) tell whether various keyboard modes are on (1) or off (0) and the lower four bits tell whether the Alt, Ctrl, or Shift key is being pressed.

The lower half of Figure 6-6 shows KB_FLAG_1, a companion byte to KB_FLAG that gives additional keyboard status information. The KEYBOARD_IO routine uses KB_FLAG_1 internally, but provides no way to read it into a register. However, KB_FLAG_1 follows KB_FLAG in memory, so to obtain KB_FLAG_1, you can read the contents of location 418H (KB_FLAG is at 417H).

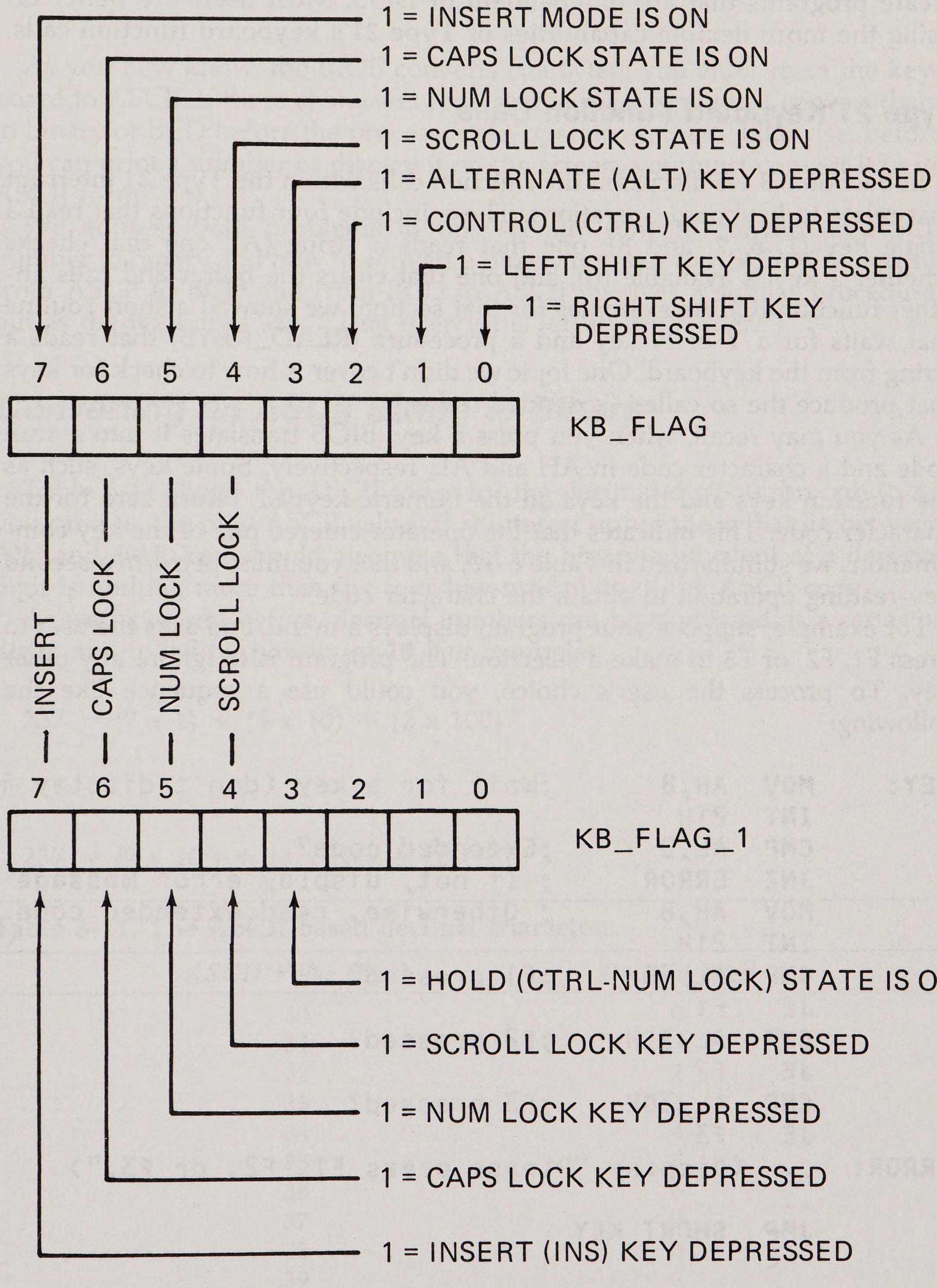


Figure 6-6. Keyboard I/O shift status bytes.

As we just mentioned, the Type 16 interrupt is the one you must use to create programs that are independent of DOS. Most users are better off using the more flexible capabilities of Type 21's keyboard function calls.

Type 21 Keyboard Function Calls

In Section 6.3 we discussed the function calls within the Type 21 interrupt that relate to keyboard operations. These include four functions that read a single key (1, 6, 7, and 8), one that reads a string (A), one that checks whether a key is available (B), and one that clears the buffer and calls another function (C). As examples for that section, we showed a short routine that waits for a Y or N key and a procedure (READ_KEYS) that reads a string from the keyboard. One topic we didn't cover is how to check for keys that produce the so-called "extended codes."

As you may recall, when you press a key, BIOS translates it into a scan code and a character code in AH and AL, respectively. Some keys, such as the function keys and the keys on the numeric keypad, return zero for the character code. This indicates that the operator entered one of the key combinations we summarized in Table 6-10, and that you must perform a second key-reading operation to obtain the character code.

For example, suppose your program displays a menu, and asks the user to press F1, F2, or F3 to make a selection. The program is to ignore any other key. To process the user's choice, you could use a sequence like the following:

```
KEY:    MOV   AH,8          ;Wait for a key (don't display it)
        INT   21H
        CMP   AL,0          ;Extended code?
        JNZ   ERROR         ; If not, display error message
        MOV   AH,8          ; Otherwise, read extended code
        INT   21H
        CMP   AL,3BH        ;F1 pressed?
        JE    F1
        CMP   AL,3CH        ;F2 pressed?
        JE    F2
        CMP   AL,3DH        ;F3 pressed?
        JE    F3
ERROR:  ..   (Display "Please press F1, F2, or F3.")
        ..
        JMP   SHORT KEY
```

6.5 ASCII/Binary Code Conversions

As you now know, the BIOS converts characters you enter from the keyboard to ASCII. If these characters represent *numbers,* you must convert them to binary or BCD before the processor can operate on them. Likewise, before you can print a number or display it on the screen, you must convert it to its ASCII form.

We address both problems in this section: how to convert an ASCII number to binary and how to convert a binary number to ASCII. (Converting from ASCII to BCD and from BCD to ASCII involves a similar procedure, but as the textbooks say, "That exercise is left to the reader.")

Converting an ASCII String to Binary

Table 6-11 shows the ASCII codes for the decimal digits 0 through 9. As you can see, the only ASCII values that interest us are those that lie between 30H and 39H. You should also note that the binary equivalent of a decimal digit is nothing more than the four low-order bits of the ASCII code.

As we have said before, decimal numbers can be expressed as a series of digits multiplied by powers of 10. For example:

$$237 = (7 \times 1) + (3 \times 10) + (2 \times 100)$$

or

$$237 = (7 \times 10^0) + (3 \times 10^1) + (2 \times 10^2)$$

Table 6-11. The ASCII-based decimal characters.

ASCII Value (Hex)	*Decimal Digit*
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9

Since you enter digits of a number one at a time, an ASCII-based decimal to binary conversion routine must include a multiply-by-10 operation. For

instance, if the operator types 93, you must multiply the 9 by 10 before adding the 3. In general, the conversion process proceeds in this order:

- For the first (most-significant) digit, the conversion routine must convert the digit to binary by stripping off the four high-order bits of the ASCII code, and store the binary value as a partial result.
- For any remaining digits, the conversion routine must convert the digit to binary, multiply the previous partial result by 10, then add the new digit to the product (thereby updating the partial result).

An ASCII-to-Binary Conversion Algorithm

You generally need to convert negative as well as positive numbers, and numbers that include a decimal point, so our conversion program must account for minus sign (-) and point (.) characters, too. Figure 6-7 is a flowchart for an algorithm to convert an ASCII string in memory into a two's-complement (signed) binary number. We assume that the number fits into 16 bits, so its limits are -32768 and +32767.

To begin, the result and decimal count (number of digits to the right of the decimal point) are set to zero, and the program scans past any leading blanks. At this point, the program takes either of two paths; one for negative numbers, the other for positive numbers.

Both paths are nearly identical, except that a converted negative number is checked against -32,768 and must be complemented, whereas the converted positive number is checked against 32,767. A procedure called CONV_AB, which is flowcharted in Figure 6-7(A), makes the actual conversion.

The CONV_AB procedure begins by checking whether the next string character is a decimal point. If it is, CONV_AB records the remaining character count as the *decimal count,* then advances the string pointer. If the next character is not a decimal point, CONV_AB checks whether it is a decimal digit. If this character is not a digit, CONV_AB declares it "invalid" and sets an error indicator, then returns to the main program.

Upon finding a valid digit character, CONV_AB multiplies the current partial result by 10, then converts the ASCII character to a digit and adds it to the result. If the addition produces a carry, CONV_AB sets the error indicator and returns. Otherwise it increments the pointer and returns to the decimal point-checking instructions. When the entire string has been converted, CONV_AB returns to the main program.

ASCII-to-Binary Conversion Program

Example 6-7 shows a procedure that implements the preceding algorithm. This procedure (ASCII_BIN) converts an ASCII string in the data segment—

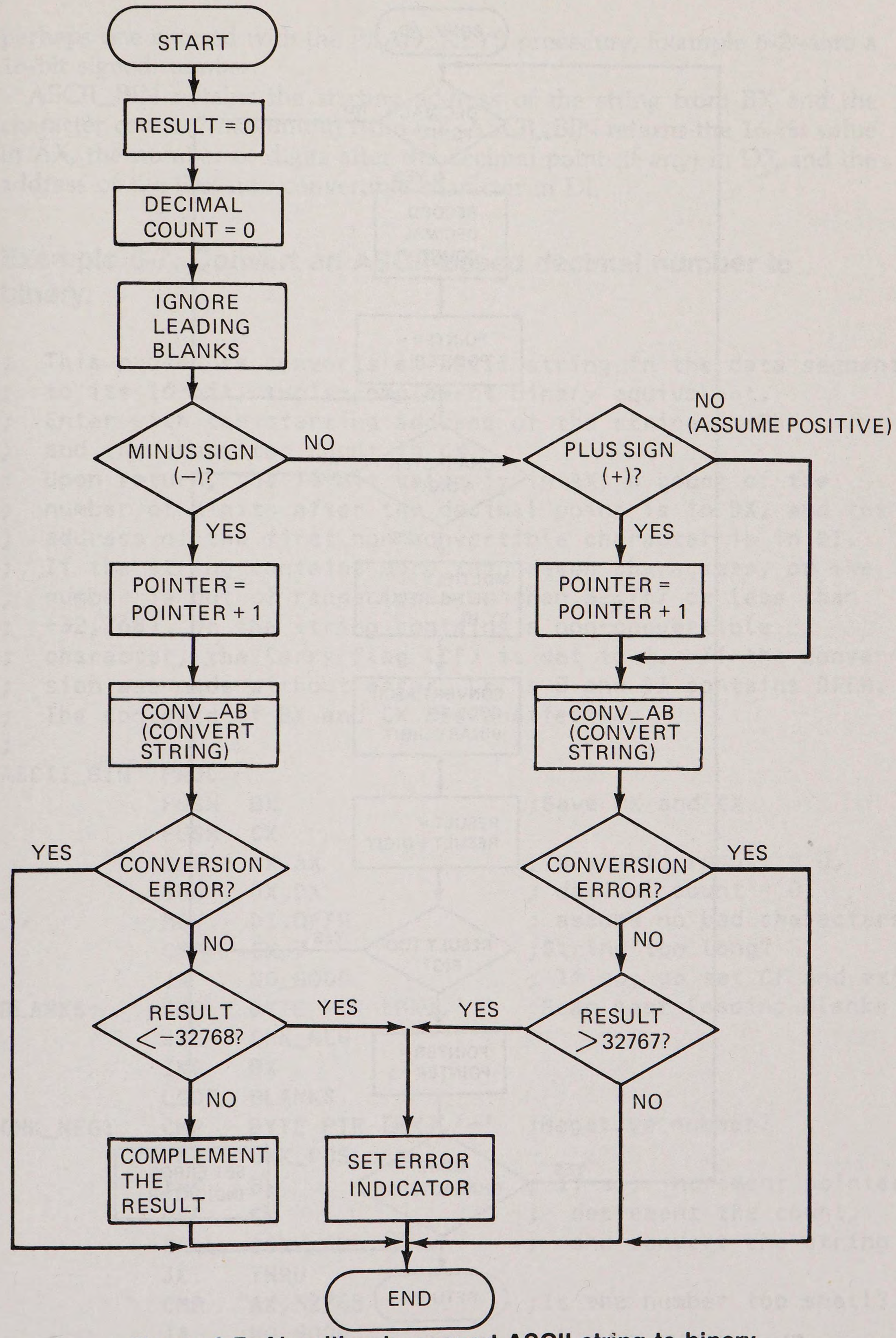


Figure 6-7. Algorithm to convert ASCII string to binary.

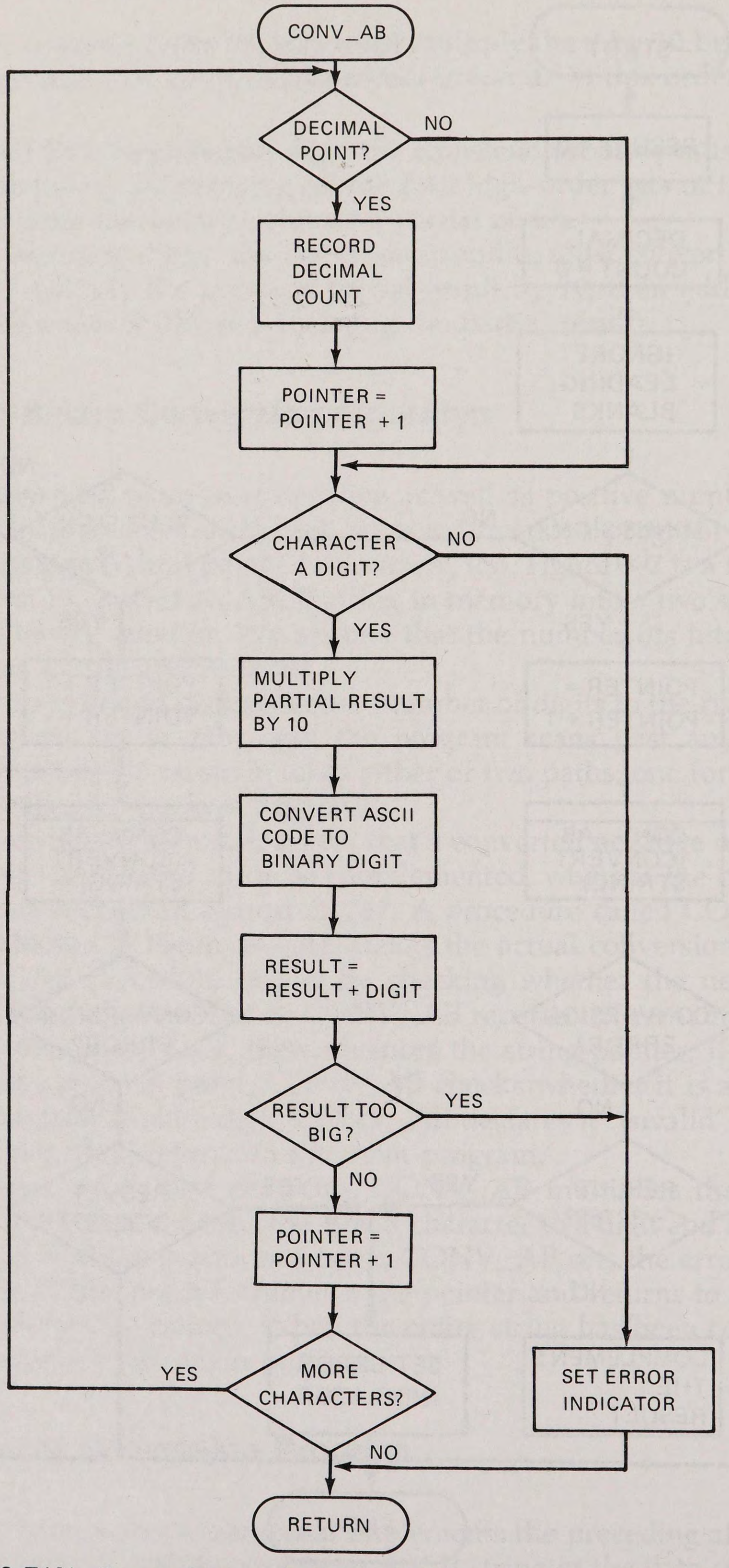


Figure 6-7(A). Procedure called by ASCII conversion algorithm.

perhaps one entered with the READ_KEYS procedure, Example 6-2—into a 16-bit signed number.

ASCII_BIN obtains the starting address of the string from BX and the character count (7 maximum) from CX. ASCII_BIN returns the 16-bit value in AX, the number of digits after the decimal point (if any) in DX and the address of the first non-convertible character in DI.

Example 6-7. Convert an ASCII-based decimal number to binary.

```
;  This procedure converts an ASCII string in the data segment
;  to its 16-bit, two's-complement binary equivalent.
;  Enter with the starting address of the string in BX
;  and the character count in CX.
;  Upon return, the 16-bit value is in AX, a count of the
;  number of digits after the decimal point is in DX, and the
;  address of the first non-convertible character is in DI.
;  If the string contains more than seven characters, or the
;  number is out of range (greater than 32,767 or less than
;  -32,768), or the string contains a non-convertible
;  character, the Carry Flag (CF) is set to 1.  If the conver-
;  sion was made without error, CF is 0 and DI contains 0FFH.
;  The contents of BX and CX are unaffected.
;
ASCII_BIN  PROC
           PUSH  BX                      ;Save BX and CX
           PUSH  CX
           SUB   AX,AX                   ;To start, result = 0,
           SUB   DX,DX                   ; decimal count = 0,
           MOV   DI,0FFH                 ; assume no bad characters
           CMP   CX,7                    ;String too long?
           JA    NO_GOOD                 ; If so, go set CF and exit
BLANKS:    CMP   BYTE PTR [BX],' '       ;Scan past leading blanks
           JNE   CHK_NEG
           INC   BX
           LOOP  BLANKS
CHK_NEG:   CMP   BYTE PTR [BX],'-'       ;Negative number?
           JNE   CHK_POS
           INC   BX                      ; If so, increment pointer,
           DEC   CX                      ;  decrement the count,
           CALL  CONV_AB                 ;  and convert the string
           JC    THRU
           CMP   AX,32768                ;Is the number too small?
           JA    NO_GOOD
```

```
                NEG     AX                      ; No.  Complement the result
                JS      GOOD
CHK_POS:        CMP     BYTE PTR [BX],'+'       ;Positive number?
                JNE     GO_CONV
                INC     BX                      ; If so, increment pointer,
                DEC     CX                      ;  decrement the count,
GO_CONV:        CALL    CONV_AB                 ;  and convert the string
                JC      THRU
                CMP     AX,32767                ;Is the number too big?
                JA      NO_GOOD
GOOD:           CLC
                JNC     THRU
NO_GOOD:        STC                             ;  If so, set Carry Flag
THRU:           POP     CX                      ;Restore registers
                POP     BX
                RET                             ; and exit
ASCII_BIN       ENDP
;
;  This procedure performs the actual conversion.
;
CONV_AB         PROC
                PUSH    BP                      ;Save scratch registers
                PUSH    BX
                PUSH    SI
                MOV     BP,BX                   ;Put pointer in BP
                SUB     BX,BX                   ; and clear BX
CHK_PT:         CMP     DX,0                    ;Decimal point already found?
                JNZ     RANGE                   ; If so, skip following check
                CMP     BYTE PTR DS:[BP],'.'    ;Decimal point?
                JNE     RANGE
                DEC     CX                      ; If so, decrement count,
                MOV     DX,CX                   ;  and record it in DX
                JZ      END_CONV                ;  Exit if CX = 0
                INC     BP                      ;  Increment pointer
RANGE:          CMP     BYTE PTR DS:[BP],'0'    ;If the character is not a
                JB      NON_DIG                 ; digit...
                CMP     BYTE PTR DS:[BP],'9'
                JBE     DIGIT
NON_DIG:        MOV     DI,BP                   ; put its address in DI,
                STC                             ; set the Carry Flag,
                JC      END_CONV                ; and exit
DIGIT:          MOV     SI,10                   ;The character is a digit,
                PUSH    DX
                MUL     SI                      ; so multiply AX by 10
                POP     DX
```

```
            MOV     BL,DS:[BP]          ; Fetch ASCII code,
            AND     BX,0FH              ; save only low bits,
            ADD     AX,BX               ; and update partial result
            JC      END_CONV            ; Exit if result is too big
            INC     BP                  ; Otherwise, increment BP
            LOOP    CHK_PT              ; and continue
            CLC                         ;When done, clear Carry Flag
END_CONV:   POP     SI                  ;Restore registers
            POP     BX
            POP     BP
            RET                                 ; and return to caller
CONV_AB     ENDP
```

The value in DX indicates the magnitude of the result. This tells you what *scale factor* to apply if you are operating on converted numbers of mixed sizes. The contents of DX can range from 0 (the result is an integer) to 5 (the result is a pure fraction). For example, if AX contains 1000H (decimal 4096) and DX contains 2, your result represents the decimal value 40.96.

If you want to add this value to a previous result that returned 3 in DX, you must first divide the previous result by 10. Similarly, to add 40.96 to a previous result that returned 0 in DX, you must first divide the new result by 100.

The Carry Flag (CF) indicates if an error occurred during the conversion. If CF is 0, the results are valid, but if CF is 1, ASCII_BIN detected one of the following errors:

- If the string was longer than seven characters (CX > 7), AX and DX hold 0 and DI holds 00FFH.
- If ASCII_BIN found an invalid character, DI holds its offset value.
- If a number was out-of-range (more negative than -32768 or more positive than 32767), AX is non-zero and DI holds 00FFH.

To check the validity of the answer, call ASCII_BIN in this context:

```
            CALL    ASCII_BIN   ;Call the conversion procedure
            JNC     VALID       ;Is the answer valid?
            OR      DI,DI       ; No.  Find the error condition
            JNZ     INV_CHAR
            OR      AX,AX
            JNZ     RANGE_ER
            ..                  ;  String was too long
            ..
RANGE_ER:   ..                  ;  Number out-of-range
            ..
INV_CHAR:   ..                  ;  Invalid character
```

```
                ..
VALID:          ..                      ;The answer is valid
                ..
```

Converting a Binary Number to ASCII

To print a result or display it on the screen, you must convert it to ASCII. Fortunately, this is easy to do. To convert a 16-bit binary number to ASCII, you need a program that determines how many 1s, 10s, 100s, 1000s, and 10000s the number contains, and that converts each of those counts into an ASCII character. You can either output the ASCII characters as they are calculated or store them in memory as a string and output them later with another program.

Example 6-8 is a procedure called BIN_ASCII that converts a 16-bit binary number in AX to an ASCII string in memory. To derive the various counts, BIN_ASCII successively divides the contents of AX by 10, then uses the remainder of each divide operation to build the string. BIN_ASCII returns the address of the converted string in BX and the character count in CX.

Example 6-8. Convert a binary number to an ASCII string.

```
;  This procedure converts a signed binary number to a six-byte
;  ASCII string (sign plus five digits) in the data segment.
;  Enter with the number to be converted in AX and the
;  starting address of the memory buffer in BX.
;  Upon return, BX holds the address of the converted output
;  string and CX holds the length of the string.
;  Other registers are preserved.
;
BIN_ASCII   PROC
            PUSH  DX            ;Save affected registers
            PUSH  SI
            PUSH  AX            ;Save binary value
            MOV   CX,6          ;Fill buffer with spaces
FILL_BUFF:  MOV   BYTE PTR [BX],' '
            INC   BX
            LOOP  FILL_BUFF
            MOV   SI,10         ;Get ready to divide by 10
            OR    AX,AX         ;If value is negative,
            JNS   CLR_DVD
            NEG   AX            ; make it positive
CLR_DVD:    SUB   DX,DX         ;Clear upper half of dividend
            DIV   SI            ;Divide AX by 10
```

```
        ADD     DX,'0'          ;Convert remainder to ASCII digit
        DEC     BX              ;Back up through buffer
        MOV     [BX],DL         ;Store this char. in the string
        INC     CX              ;Count converted character
        OR      AX,AX           ;All done?
        JNZ     CLR_DVD         ; No.  Get next digit
        POP     AX              ; Yes.  Retrieve original value
        OR      AX,AX           ;Was it negative?
        JNS     NO_MORE
        DEC     BX              ; Yes.  Store sign
        MOV     BYTE PTR [BX],'-'
        INC     CX              ;  and increase character count
NO_MORE:POP     SI              ;Restore registers
        POP     DX
        RET                     ; and exit
BIN_ASCII ENDP
```

Study Exercises (answers on page 365)

1. Develop a procedure to calculate the elapsed time between two key depressions. Set the time to zero at the beginning of the timing operation; the elapsed time will be the current time when the second key is pressed.
2. To find out the *minimum* time between keystrokes, execute the procedure in Step 1 and, when you see the cursor, press the space bar twice, quickly.
3. I wrote a program that is supposed to display the messages *Sort Program* and *Press any key to begin.* on separate lines, but instead, it produces

```
Sort ProgramPress any key to begin.
```

   Why does it do this?
4. Write a program to display a message of the form

```
Try again. You have n starfighters left.
```

   where n is a number between 1 and 6 in CX.

7

Graphics Made Easy

7.1 Display Modes

If you have standard IBM products, your display connects to the PC through either of two adapter cards. The *Monochrome Display and Parallel Printer Adapter* connects to both the IBM Monochrome Display and a parallel printer. The *Color/Graphics Monitor Adapter* connects to a monitor or a standard TV set.

Monochrome Adapter

The Monochrome Display and Parallel Printer Adapter can display only black and white *alphanumeric* characters—letters, numbers, and symbols—and block graphics characters. The adapter derives these characters by translating each of the 256 ASCII character codes into a display character. The adapter displays an 80-column by 25-line character grid on the screen based on the data in an on-card 4K-byte buffer.

Color/Graphics Adapter

The Color/Graphics Monitor Adapter can operate in two modes, Alphanumeric (just discussed) or Graphics, and can display in black and white or color. It provides 16 colors.

In the Alphanumeric mode, the adapter can generate an 80x25 character high-resolution display or a 40x25 character low-resolution display. The adapter card contains 16K bytes of memory, and can store up to four pages of 80x25 screen information or up to eight pages of 40x25 screen information.

In the Graphics mode the adapter divides the screen into a grid of dot-like picture elements, or *pels,* rather like a scoreboard or a time/temperature dis-

play. Resolutions of 320x200 pels and 640x200 pels are available. With 320x200 resolution, each pel may have one of four colors and the background may have any of 16 possible colors. The 640x200 resolution mode displays only black and white because the adapter needs the full 16K bytes of on-card storage to define the on or off states of the pels.

Due to the large number of pels and a palette of 16 colors, programming displays in the Graphics mode can be a somewhat complex task. (For more on Graphics mode graphics, refer to the IBM *Technical Reference* manual.) For this reason, we limit our discussion to programming in the Alphanumeric mode, using the block graphics characters in the character set. Further, all procedures in this chapter display in black and white on an 80x25 screen grid. You can easily modify them for color and/or the 40x25 grid, however.

7.2 Displaying Characters

The Character Set

As we mentioned in Section 7.1, when the display adapter is operating in the Alphanumeric mode (as it will be unless you've made a program switch to Graphics mode), it forms characters by converting ASCII codes in its on-card buffer into any of 256 display characters. Table 7-1 shows the relationship between the ASCII codes and the display characters.

Note that besides the standard letters, numbers, and symbols, the character set includes foreign language letters, Greek and math symbols, arrows, and a variety of other "goodies." For graphics applications, the characters that interest us most are those in columns 0, 1, A, B, and C. Admittedly, these simple block graphics characters do not let you create complex graphic displays that compete with Disney or George Lucas, but they are convenient for making sales demonstration programs, games, and flashy business programs.

Attributes

Besides an eight-bit code, every character position on the screen also has an eight-bit *attribute*. The attribute specifies the colors of the character and its background, the intensity of the colors, and whether the character is displayed continuously or blinking.

When you turn the computer on, the adapter card sets the entire screen to a normal intensity, non-blinking display, with white characters on a black background. By specifying new attribute values, however, you can change these settings for any selected position(s) on the screen.

Table 7-1. ASCII codes for display characters.

HIGH PART

DECIMAL VALUE ➡		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
⬇	HEXA-DECIMAL VALUE	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	BLANK (NULL)	►	BLANK (SPACE)	0	@	P	‘	p	Ç	É	á	1/4 Dots On			∝	≡
1	1	☺	◄	!	1	A	Q	a	q	ü	Æ	í	1/2 Dots On			β	±
2	2	☻	↕	"	2	B	R	b	r	é	FE	ó	3/4 Dots On			γ	≥
3	3	♥	!!	#	3	C	S	c	s	â	ô	ú				π	≤
4	4	♦	¶	$	4	D	T	d	t	ä	ö	ñ				Σ	∫
5	5	♣	§	%	5	E	U	e	u	à	ò	Ñ				σ	⌠
6	6	♠	▬	&	6	F	V	f	v	å	û	ª				μ	÷
7	7	BEEP •	↨	'	7	G	W	g	w	ç	ù	º				τ	≈
8	8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿				Φ	°
9	9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐				Θ	•
10	A	◙	→	*	:	J	Z	j	z	è	Ü	¬				Ω	·
11	B	♂	ESC ←	+	;	K	[	k	{	ï	¢	½				δ	√
12	C	♀	∟	,	<	L	\	l	¦	î	£	¼				∞	η
13	D	♪	↔	–	=	M	]	m	}	ì	¥	¡				∅	2
14	E	♫	▲	.	>	N	^	n	~	Ä	Pts	«				∈	■
15	F	☼	▼	/	?	O	_	o	△	Å	f	»				∩	BLANK 'FF'

LOW PART

Table 7-2. Attribute byte for black and white display.

Video Setting	*Bit Position 7 6 5 4 3 2 1 0*	*Character Color*	*Background Color*
Normal	B 0 0 0 I 1 1 1	White	Black
Reverse Video	B 1 1 1 I 0 0 0	Black	White
Non-Display (Black)	B 0 0 0 I 0 0 0	White	Black
Non-Display (White)	B 1 1 1 I 1 1 1	White	Black

I = 0 normal intensity
= 1 normal intensity
B = 0 non-blinking
= 1 blinking foreground

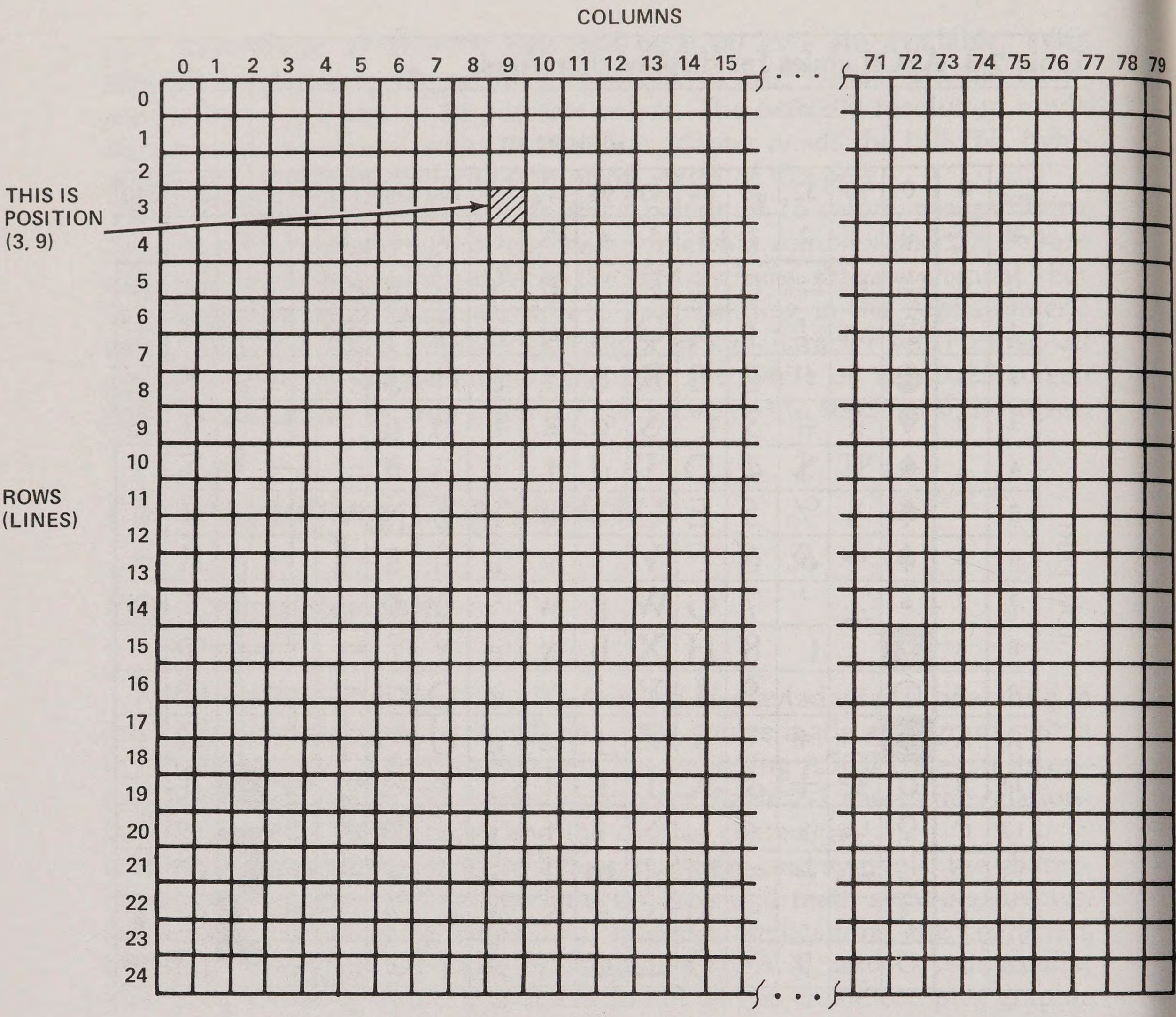


Figure 7-1. 80x25 Alphanumeric screen grid.

Table 7-2 shows the formats of the attribute byte for displaying characters in black and white. Chapter 2 of the IBM *Technical Reference* manual describes the attribute formats for displaying characters in color.

Display Commands

As we mentioned in Section 7.1, all procedures in this chapter display characters on an 80-column by 25-line grid, in black and white. On the 80x25 grid, each character position has a vertical (row or line) coordinate between 0 and 24, and a horizontal (column) coordinate between 0 and 79. Therefore, if you wish to display a character at the tenth position on the

fourth line, you must specify a row coordinate of 3 and a column coordinate of 9. For convenience we can use the notation (3,9). Figure 7-1 shows this grid position.

How can we communicate with the screen? We have already seen the programs that do this. They are the options provided by the Type 10 (Video I/O) interrupt, which were described in Section 6.2 and summarized in Table 6-2. For your convenience, Table 7-3 shows a subset of Table 6-2 that lists only the options that pertain to 80x25, Alphanumeric, black and white display operations. If you are using a Monochrome Display, which has only one display page and one video mode (80x25, black and white, Alphanumeric), disregard the commands and inputs printed in italic type.

Incidentally, note that Table 7-3 includes two separate write (output) options. Option (AH) = 9 outputs a character and an attribute to the cursor location, whereas (AH) = 10 outputs just a character and gives it whatever attribute you previously assigned to that grid position. The second option is especially convenient with black and white displays, where you rarely change the attribute.

A Simple Display Operation

We can now begin generating some Alphanumeric graphics. To start, let's display a low-resolution diagonal "line" on the screen. This line starts at the top left-hand corner and runs diagonally down the screen, advancing one row and one column at a time.

Let's use the "smile face" character (code 02 in Table 7-1) to form the line. Thus, the first face is at screen position (0,0), the next face is at (1,1), and so on.

Example 7-1 shows a procedure (DIAG_LINE) that plots this diagonal line. DIAG_LINE makes use of four of the Type 10 interrupt options: Option (AH) = 15 reads the display page number into BH; (AH) = 0 selects the 80x25, black and white, and Alphanumeric modes; (AH) = 2 positions the cursor; and (AH) = 10 sends the character to the screen.

7.3 The Basics of Animation

The procedure in Example 7-1 produces a static display, one that does not move. The faces just sit there and smile at you (probably saying, "Have a nice day.") In many applications, however, you'd like to move the shapes on the screen. That is, you'd like to *animate* them.

Producing animated affects is not as difficult as you might expect. In fact, it requires only five steps:

Table 7-3. Type 10 Interrupt options for 80×25, black and white, and Alphanumeric.

(AH)	*Operation*	*Additional Input Registers*	*Result Registers**
	CRT Interface Routines		
0	*Set video mode*	*(AL) = 2 80×25, black and white, and alphanumeric*	*None*
2	Set cursor position	(DH,DL) = Row,column (0-24,0-79) *(BH) = Page number (0-4)*	None
3	Read cursor position	*(BH) = Page number (0-4)*	(DH,DL) = Row,column of cursor (CH,CL) = Current cursor mode
5	*Select active display page*	*(AL) = New page value (0-3)*	*None*
6	Scroll active page up	(AL) = Number of lines. Input lines blanked at bottom of window. (AL) = 0 blanks entire window. (CH,CL) = Row,column of upper left corner of scroll (DH,DL) = Row,column of lower right corner of scroll (BH) = Attribute to be used on blank line	None
7	Scroll active page down	Same as above, but input lines are blanked at top of window.	None

Character-Handling Routines

8	Read attribute/ character at current cursor position	*(BH) = Display page (0-3)*	(AL) = Character read (AH) = Attribute of character read
9	Write attribute/ character at current cursor position	*(BH) = Display page (0-3)* (BL) = Attribute of character (CX) = Count of characters to write (AL) = Character to write	None
10	Write character only at current cursor position	*(BH) = Display page (0-3)* (CX) = Count of characters to write (AL) = Character to write	None

ASCII Teletype Routine for Output

14	Write character to screen, then advance cursor	(AL) = Character to write *(BH) = Display page (0-3)*	None

Read Video State

15	Read current video state	None	(AL) = Current mode. (See (AH) = 0 for explanation (AH) = Number of character columns on screen (BH) = Active display page

Note: Besides the registers listed here, these routines preserve CS, SS, DS, ES, BX, CX, and DX. All other registers should be considered destroyed.

1. Plot the shape(s) on the screen.
2. Wait a period of time, so the shape is visible.
3. Erase the shape.
4. Change the shape's row and column coordinates.
5. Repeat the process, starting at Step 1.

Example 7-1. Display a diagonal line.

```
;  This procedure draws a diagonal line of "smile faces,"
;  starting at the upper left corner of the screen.
;  All registers are preserved.
;
DIAG_LINE  PROC
           PUSH  AX          ;Save registers
           PUSH  BX
           PUSH  CX
           PUSH  DX
           MOV   AH,15       ;Set BH to active display page
           INT   10H
           MOV   AH,2        ;Select 80x25, B/W, Alphanumeric
           INT   10H
           MOV   CX,1        ;Character count = 1
           MOV   DX,0        ;Start at row = 0, column = 0
SET_CRSR:  MOV   AH,2        ;Move cursor to next position
           INT   10H
           MOV   AL,2        ;Display character is smile face
           MOV   AH,10       ;Write character to the screen
           INT   10H
           INC   DH          ;Point to next row
           INC   DL          ; and column position
           CMP   DH,25       ;Bottom of screen?
           JNE   SET_CRSR
           POP   DX          ; Yes.  Restore registers
           POP   CX
           POP   BX
           POP   AX
           RET               ;  and exit
DIAG_LINE  ENDP
```

You can generate a time delay (Step 2) by using the DELAY procedure we developed in Example 6-2.

To erase a shape from the screen you can either *clear* that portion of the screen, by writing blank (code 0) characters, or *replot* the shape in black. To replot in black, you can either write a blank character with the (AH) = 9

option or give the character a non-display attribute, (BL) = 0, with the (AH) = 10 option.

The Old Moving Face Trick

Example 7-2 shows a new line-plotting procedure that "moves" the smile face down the diagonal in half-second spurts. The main procedure (MOVE_FACE) has only four more instructions than the DIAG_LINE procedure of Example 7-1. These call a half-second delay procedure (DLY_HALF) and erase the face by sending a blank (0) code to the screen.

You might like to experiment with longer or shorter delays. Or, to see (or *not see*) the face travel at maximum speed, you can eliminate the delay entirely.

Example 7-2. Animate a diagonal line display.

```
;  This procedure moves a "smile face" diagonally down the
;  screen, starting at the upper left corner, with a 1/2-second
;  delay between moves.  All registers are preserved.
;
            EXTRN  DELAY:FAR
MOVE_FACE   PROC
            PUSH   AX          ;Save registers
            PUSH   BX
            PUSH   CX
            PUSH   DX
            MOV    AH,15       ;Set BH to active display page
            INT    10H
            MOV    AH,2        ;Select 80x25, B/W, Alphanumeric
            INT    10H
            MOV    CX,1        ;Character count = 1
            MOV    DX,0        ;Start at row = 0, column = 0
SET_CRSR:   MOV    AH,2        ;Move cursor to next position
            INT    10H
            MOV    AL,2        ;Display character is smile face
            MOV    AH,10       ;Write character to the screen
            INT    10H
            CALL   DLY_HALF    ;Wait a half-second,
            SUB    AL,AL       ; then erase the face
            MOV    AH,10
            INT    10H
            INC    DH          ;Point to next row
            INC    DL          ; and column position
```

```
            CMP     DH,25       ;Bottom of screen?
            JNE     SET_CRSR
            POP     DX          ; Yes.  Restore registers
            POP     CX
            POP     BX
            POP     AX
            RET                 ;  and exit
MOVE_FACE   ENDP
;
;  This procedure generates a one-half second delay.
;
DLY_HALF    PROC
            SUB     AL,AL       ;Set minutes to zero
            SUB     BH,BH       ;Set seconds to zero
            MOV     BL,50       ;Set hundredths to 50 (1/2 sec.)
            CALL    DELAY
            RET
DLY_HALF    ENDP
```

7.4 Creating Complex Shapes with Shape Tables

Shapes that are formed from several characters are almost as easy to manipulate as those formed from a single character, like our smile face. But multi-character shapes require multiple write operations—one for each character in the shape.

If a shape consists of just two or three characters, and you want to display only that shape, you can probably program the job with a few successive output operations. However, if a shape contains many characters, or if you want to display two or more shapes simultaneously, consider putting the shape parameters in a *shape table*.

A shape table is a data table that holds the character code, attribute, and row and column offset for each character in a display shape. You already know what the character code and attribute are. The row and column offsets give the vertical and horizontal distances between the previous character and the current character. In this way, the offsets "tell" the computer how to draw the shape.

To illustrate, Figure 7-2 shows a "car" constructed with seven Alphanumeric characters. Moving left to right, the body consists of the letter R (code 52H), a 1/2-dots-on character (B1), and two solid boxes (DB); the first two characters are in reverse video, the second two are in normal video. The wheels, which are letter Os (4F), and the smile face (02) are in normal video. This shape was designed by my son, Ryan.

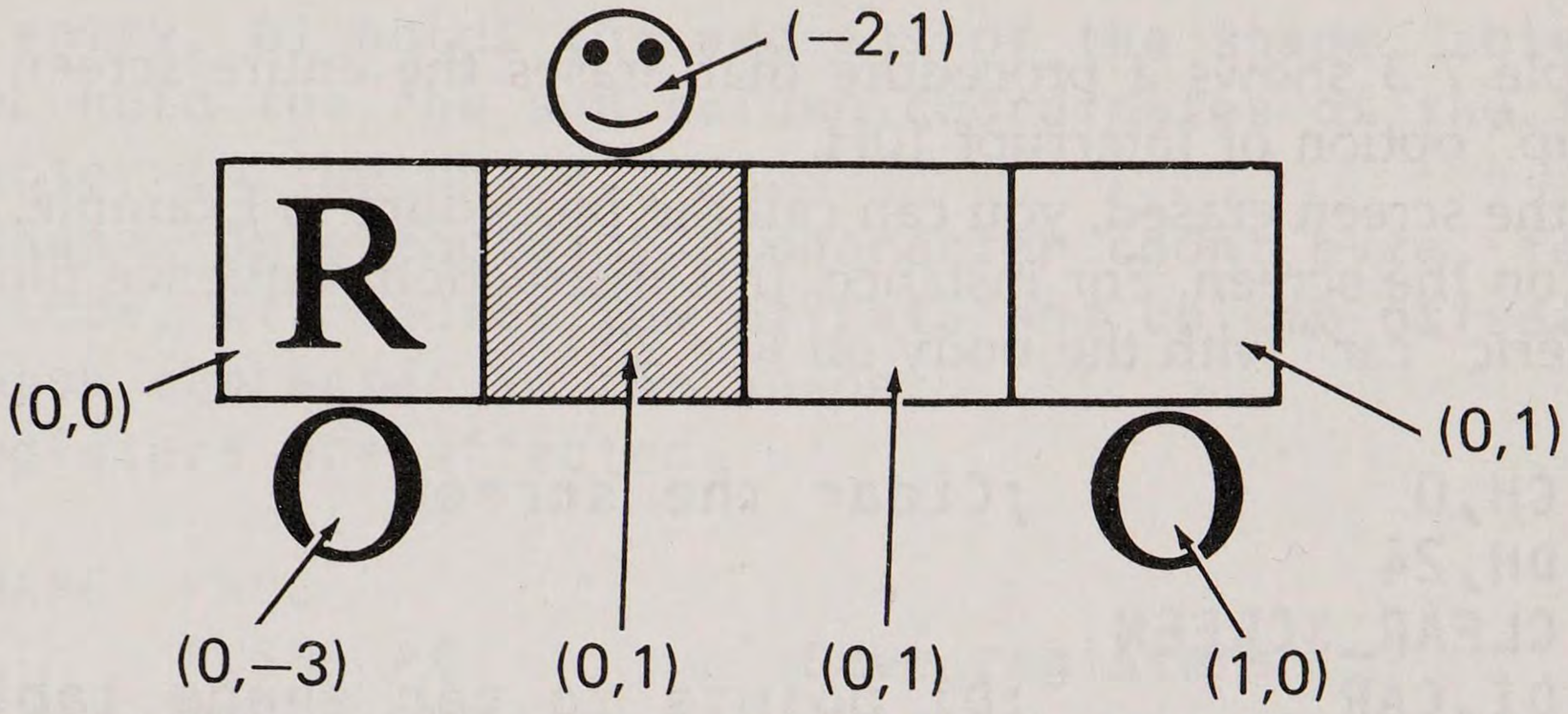


Figure 7-2. Row and column offsets for an Alphanumeric "car."

The parenthetical numbers in Figure 7-2 are the row and column offsets. The first character we plot is the letter R, so its offsets are (0,0). Next we plot the 1/2-dots-on character. It is on the same row as the R, but one column to the right of the R, so its offsets are (0,1). Similarly, the front characters of the body also have the offsets (0,1).

Next we draw the front wheel, which has offsets of (1,0) from the rightmost character of the body, and then the back wheel, which has offsets of (0,-3) from the front wheel. Finally, the smile face has offsets of (-2,1) from the back wheel.

Each shape parameter is a byte value, so the shape table must contain four bytes for each character. If you are operating on more than one shape table, each shape table should also include a byte that contains the character count. Figure 7-3 shows the shape table for the "car" we just described.

```
CAR   DB   7                  ;Shape contains seven characters
      DB   52H,70H,0,0        ;Rear character of body
      DB   0B1H,70H,0,1       ;Driver's compartment
      DB   0DBH,7,0,1         ;Next character of body
      DB   0DBH,7,0,1         ;Front character of body
      DB   4FH,7,1,0          ;Front wheel
      DB   4FH,7,0,-3         ;Rear wheel
      DB   2,7,-2,1           ;Smile face
```

Figure 7-3. Shape table for the Alphanumeric "car."

General Display Procedure

Your display program can consist of a routine that loads shape table parameters into the appropriate registers, then issues the output command. You should put this routine in a loop, where it repeatedly executes until every character in the shape table has been output.

Example 7-3 shows a procedure that erases the entire screen using the "scroll-up" option of interrupt 10H.

With the screen erased, you can call the procedure in Example 7-4 to plot a shape on the screen. For instance, this instruction sequence plots our Alphanumeric "car" with the body on line 20:

```
MOV     CH,0                ;Clear the screen
MOV     DH,24
CALL    CLEAR_SCREEN
LEA     DI,CAR              ;DI points to car shape table
MOV     DH,20               ;Body is on line 20,
MOV     DL,0                ; starting in column 0
CALL    DSPLY_SHAPE         ;Draw the car
```

Example 7-3. Clear the screen.

```
;  This procedure clears the screen by giving every line
;  the normal white-on-black attribute.
;  All registers are preserved.
;
CLEAR_SCREEN  PROC
              PUSH  AX      ;Save registers
              PUSH  BX
              PUSH  CX
              PUSH  DX
              MOV   AH,6    ;Clear the screen with the
              MOV   AL,0    ; scroll-up option.
              MOV   CX,0    ;Clear from row 0 and column 0 to
              MOV   DH,24   ; row 24
              MOV   DL,79   ; and column 79.
              MOV   BH,7
              INT   10H
              POP   DX      ;Restore registers
              POP   CX
              POP   BX
              POP   AX
              RET           ; and return
CLEAR_SCREEN  ENDP1
```

Example 7-4. Construct a shape from a shape table.

```
;  This procedure constructs a shape on the screen based on
;  a shape table in the data segment.
```

```
;  Upon entry, DI holds the address of the shape table and DH
;  and DL hold the row and column coordinates of the first
;  character in the shape.
;  The shape table contains a character count byte, followed
;  by a code, attribute, row offset, and column offset byte
;  for each character in the shape.
;  No registers are affected.
;
DSPLY_SHAPE  PROC
             PUSH  AX            ;Save registers
             PUSH  BX
             PUSH  CX
             PUSH  DX
             PUSH  DI
             MOV   AH,15         ;Set BH to active display page
             INT   10H
             SUB   CH,CH         ;Clear high byte of count
             MOV   CL,[DI]       ;CL holds char. count
             INC   DI            ;DI points to first char.
NEXT_CHAR:   ADD   DH,[DI+2]     ;Update row pointer
             ADD   DL,[DI+3]     ; and column pointer
             MOV   AH,2          ;Move cursor
             INT   10H
             MOV   AL,[DI]       ;Fetch char. value
             MOV   BL,[DI+1]     ; and attribute
             PUSH  CX            ;Save character count
             MOV   CX,1          ;Write character to screen
             MOV   AH,9
             INT   10H
             POP   CX            ;Restore character count
             ADD   DI,4          ;DI points to next character block
             LOOP  NEXT_CHAR     ;When all characters are displayed,
             POP   DI            ; restore registers
             POP   DX
             POP   CX
             POP   BX
             POP   AX
             RET                 ; and exit
DSPLY_SHAPE  ENDP
```

A Mover-and-Shaper

You can animate a complex shape, such as our "car," the same way you animate a single-character shape. To make a multi-character shape move on

the screen, you plot the shape, wait for some time interval to elapse, erase the shape, then redraw it in a new location.

As we mentioned earlier, to erase a shape, you can either clear the part of the screen where the shape is displayed or redraw the shape in black. But be aware that if there are several shapes on the screen, you must somehow keep track of where the shape to be erased is located at any given time. This might be a problem if the shape is moving.

Example 7-5 shows a procedure that moves a shape from left to right across the screen. The main procedure, MOVE_SHAPE, plots the shape on the screen using the same general approach as DSPLY_SHAPE. The delay period (1/4 second, in this case) and the screen erase function are provided by two additional procedures: DLY_QRTR and ERASE.

Here, ERASE erases the shape by replotting it with an attribute of zero. This produces a black-on-black display (that is, a *non-display*). ERASE also increments the column number—the variable COL_NO in the data segment—at the end of the erase operation. Since the rightmost column on the screen is Column 79, MOVE_SHAPE terminates when any character in the shape has a column number greater than or equal to 80.

Example 7-5. Animate a multi-character shape.

```
;  This procedure moves a shape horizontally across the screen,
;  with a 1/4-second delay between moves.
;  Upon entry, DI holds the address of the shape table, and DH
;  and DL hold the row and column coordinates of the first
;  character in the shape.
;  No registers are affected.
;
;  Set up these temporary storage locs. in the data segment.
;
CHAR_CNT   DW  ?
POINTER    DW  ?
LINE_NO    DB  ?
COL_NO     DB  ?
;
;  This is the main procedure.
;
           EXTRN  DELAY:FAR
MOVE_SHAPE PROC
           PUSH   AX           ;Save registers
           PUSH   BX
           PUSH   CX
           PUSH   DX
           PUSH   DI
```

```
        MOV     AH,15           ;Set BH to active display page
        INT     10H
        SUB     CH,CH           ;Clear high byte of count
        MOV     CL,[DI]         ;CL holds char. count
        INC     DI              ;DI points to first char.
        MOV     CHAR_CNT,CX     ;Save character count,
        MOV     POINTER,DI      ; shape table pointer
        MOV     LINE_NO,DH      ; and coordinates
        MOV     COL_NO,DL
;
;  These instructions plot the shape on the screen.
;
PLOT_NEXT:  ADD     DH,[DI+2]   ;Update row pointer
        ADD     DL,[DI+3]       ; and column pointer
        CMP     DL,80           ;Is this char. off the screen?
        JB      MOV_CRSR
        CALL    ERASE           ; If so, erase the shape and exit
        POP     DI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
MOV_CRSR:   MOV     AH,2        ; Otherwise, move cursor
        INT     10H
        MOV     AL,[DI]         ;Fetch char. value
        MOV     BL,[DI+1]       ; and attribute
        PUSH    CX              ;Save character count
        MOV     CX,1            ;Write character to screen
        MOV     AH,9
        INT     10H
        POP     CX              ;Restore character count
        ADD     DI,4            ;DI points to next character block
        LOOP    PLOT_NEXT       ;When all characters are displayed,
        CALL    DLY_QRTR        ; wait a quarter-second
        CALL    ERASE           ; then erase the shape
        JMP     SHORT PLOT_NEXT
MOVE_SHAPE  ENDP
;
;  This procedure erases a shape by replotting it with
;  attribute = 0.
;
ERASE       PROC
        MOV     CX,CHAR_CNT     ;Reload character count,
        MOV     DI,POINTER      ; shape table pointer,
```

```
                MOV     DH, LINE_NO   ; line number,
                MOV     DL,COL_NO     ; and column number
ERASE_NEXT:     ADD     DH,[DI+2]
                ADD     DL,[DI+3]
                MOV     AH,2
                INT     10H
                MOV     AL,[DI]       ;To erase a character, use an
                MOV     BL,0          ; attribute of 0
                PUSH    CX
                MOV     CX,1
                MOV     AH,9
                INT     10H
                POP     CX
                ADD     DI,4
                LOOP    ERASE_NEXT
                MOV     CX,CHAR_CNT ;Reload character count,
                MOV     DI,POINTER  ; shape table pointer,
                MOV     DH,LINE_NO  ; and line number
                INC     COL_NO      ;Point to next column
                MOV     DL,COL_NO
                RET                 ; and return to caller
ERASE           ENDP
;
;  This procedure generates a one-quarter second delay.
;
DLY_QRTR        PROC
                SUB     AL,AL       ;Set minutes to zero
                SUB     BH,BH       ;Set seconds to zero
                MOV     BL,25       ;Set hundredths to 25 (1/4 sec.)
                CALL    DELAY
                RET
DLY_QRTR        ENDP
```

Although this particular procedure only moves a shape horizontally, you can easily modify it to make a vertical or diagonal move. Just change the row coordinate (rather than the column coordinate) or both the row and the column coordinates after each display-delay-erase sequence.

Study Exercises (answers on page 366)

1. In addition to moving a shape on the screen, you can introduce another element of animation by having the shape *change* as it moves. To discover how to do this, develop a procedure that moves a "bird" across the screen, and changes the bird's shape from a lowercase "v" character

(code 76H in Table 7-1) to a "dash" character (code 0C4H), or vice versa, with each new column. Start the bird at row 20, column 0, and display it for one-tenth of a second each time it moves.

2. Game programs often use random numbers to move shapes around on the screen. To introduce yourself to this topic, develop a program that displays a "smile face" (character code 2) at column 0 of a randomly-derived line number, then moves the face across the screen.

   The face should always advance one column at a time, but may move one line lower or one line higher, depending on whether a random number generator produces 0 (down one line), 1 (up one line) or 2 (same line).

   The display operation should end when the face crosses line 0, line 24, or column 79. Use the Type (Time of Day) 1A interrupt to derive both the initial line number and the up/down/same indicator; see Example 6-1 in Section 6.2.

8

Let There Be Sound!

The PC's BASIC provides a SOUND statement that lets you generate sound through the internal speaker. SOUND takes two operands: frequency and duration. The frequency can range from 37 to 32767 Hertz (cycles/second); the duration from 0 to 65535 "clock ticks." (Clock ticks are the 8253 Timer interrupts we discussed in Chapter 6. Recall that they occur about 18.2 times a second.)

In this chapter we develop an assembly language counterpart of SOUND, and use it to play music through the speaker. Our assembly language procedure is somewhat more flexible, however, in that it lets you enter durations in increments of 1/100 of a second instead of the 1/20-second increments BASIC's SOUND provides.

8.1 How the Speaker Works

Most I/O operations in the PC are regulated by an *8255 Programmable Peripheral Interface (PPI)* chip on the System Board. The PPI contains three 8-bit registers; two for input functions and one for output functions. The input registers are assigned I/O port numbers 60H and 62H; the output register is assigned I/O port number 61H.

As Figure 8-1 shows, either or both of two sources can drive the speaker, as selected by two bits in the PPI's output register. When bit 0 is 1, the 8253 Timer chip drives the speaker. When bit 1 is 1, the speaker goes on, and stays on until bit 1 becomes 0. The programs in this chapter use the 8253 Timer technique.

8.2 Programming the Speaker

To avoid reinventing the wheel, we have modeled our sound procedure after one that the BIOS uses to beep the speaker if its power-up reliability

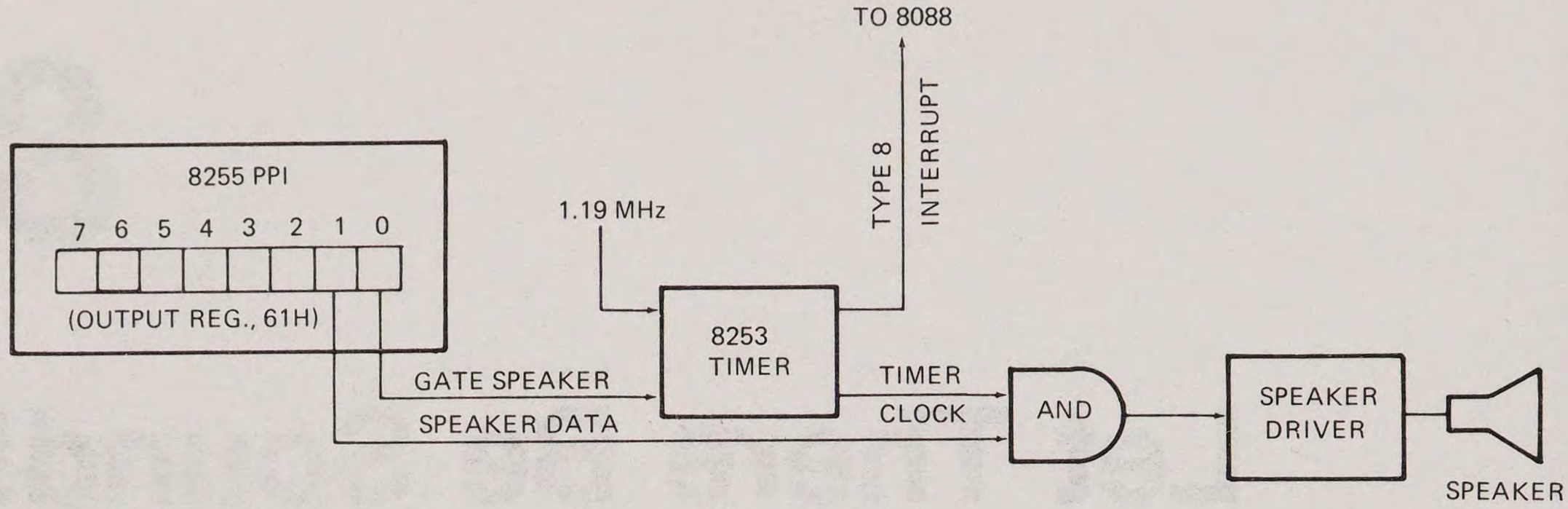


Figure 8-1. Speaker drive system.

tests find a hardware error in the system. This procedure, called BEEP, starts at offset E666 in the PC's BIOS and at offset FA08 in the XT's BIOS.

The BIOS "BEEP" Procedure

BEEP generates a 1000-Hz tone for one or more half-second intervals, based on an interval count in BX. The interval count (1 for a "short beep", 6 for a "long beep") is set by the calling procedure, ERROR_BEEP.

BEEP configures the 8253 Timer to generate a frequency output, then sends a "divisor" value of 533H to the 8253's Timer 2. This selects a frequency of 1000 Hz. After that, BEEP forces Bits 0 and 1 of AL to be 1 and outputs AL to the 8255's output register (I/O port 61H), which turns on the speaker. A subsequent loop keeps the speaker on for one or more half-second intervals, based on the value in BX.

A More General Sound Generator

BEEP is a fairly good model for a general sound-generating procedure, but we need to change two aspects of its design. First, BEEP can only produce 1000-Hz tones; we want to be able to produce tones of any frequency. Second, BEEP's durations are multiples of 1/2 seconds; we want durations that are multiples of 10 milliseconds.

The BEEP procedure shows us that sending a value of 533H (decimal 1331) to Timer 2 makes it generate a 1000-Hz tone. Therefore, to generate a tone of any other frequency we must multiply 1331 by 1000/Frequency. If our sound procedure accepts a frequency in DI, the following instructions produce the correct value for Timer 2 in AX:

```
MOV   DX,14H      ;Timer divisor =
MOV   AX,4F38H    ; 1331000/Frequency
DIV   DI
```

You can generate a 10-millisecond delay with two instructions of the general form

```
          MOV   CX,n
SPKR_ON:  LOOP  SPKR_ON
```

where *n* is the number that makes the two instructions execute in 10 milliseconds.

Appendix C shows that the MOV instruction executes in four clock cycles and the LOOP instruction executes in 17 clock cycles if the transfer takes place or five clock cycles otherwise. Since the transfer takes place until CX is 1—that is, *n-1* times—the general equation is:

$$[17(n-1) + 5 + 4][210 \times 10^{-9}] = 0.01$$

which solves to n = 2801. Hence, the instruction pair that generates a 10-millisecond delay is:

```
          MOV   CX,2801
SPKR_ON:  LOOP  SPKR_ON
```

By making these changes to BEEP, we can write a general-purpose procedure that produces sounds of any frequency (as specified by DI) and any duration, in increments of 0.01 seconds (as specified by BX). Example 8-1 shows this assembly language SOUND procedure.

As we noted in the prelude to this procedure, SOUND can generate tones between 21 and 65535 Hz. (The lower limit, 21 Hz, is the smallest value of DI that can be divided into 1,331,000 without producing an overflow.) The upper limit is, of course, superfluous, because human hearing extends to only about 20,000 Hz. Still, these limits better the BASIC SOUND statement's range of 37 to 32767 Hz.

Further, you can give SOUND duration values between 0 and 65535. Because BX gets decremented before it is tested for zero, the 0 input produces a duration of 256. Hence, SOUND can produce durations from 0.01 seconds (BX = 1) to 655.36 seconds (BX = 0).

8.3 Music, Music, Music

We can now build on our general-purpose sound procedure to play tunes through the speaker. To do this, however, we must know how frequencies relate to musical notes.

Figure 8-2 shows a portion of a piano keyboard spanning two *octaves* (an octave is eight notes) and the frequency of each note in Hertz. The lower

Example 8-1. Speaker-beeper procedure.

```
;  This procedure produces a tone of a specified frequency and
;  duration on the speaker.
;  Enter with the frequency, in Hertz, in DI (21 to 65535 Hz)
;  and the duration, in hundredths of a second, in BX (0 to
;  65535).
;  All registers are preserved.
;
SOUND       PROC
            PUSH  AX          ;Save registers
            PUSH  BX
            PUSH  CX
            PUSH  DX
            PUSH  DI
            MOV   AL,0B6H     ;Write timer mode register
            OUT   43H,AL
            MOV   DX,14H      ;Timer divisor =
            MOV   AX,4F38H    ; 1331000/Frequency
            DIV   DI
            OUT   42H,AL      ;Write Timer 2 count low byte
            MOV   AL,AH
            OUT   AL,AH       ;Write Timer 2 count high byte
            IN    AL,61H      ;Get current Port B setting
            MOV   AH,AL       ; and save it in AH
            OR    AL,3        ;Turn speaker on
            OUT   61H,AL
WAIT:       MOV   CX,2801     ;Wait 10 milliseconds
SPKR_ON:    LOOP  SPKR_ON
            DEC   BX          ;Speaker-on count expired?
            JNZ   WAIT        ; If not, keep speaker on
            MOV   AL,AH       ; Otherwise, recover value of port
            OUT   61H,AL
            POP   DI          ;Restore registers
            POP   DX
            POP   CX
            POP   BX
            POP   AX
            RET               ; and exit
SOUND       ENDP
```

octave runs from low C to middle C, while the upper octave runs from middle C to high C.

In this drawing we have labeled each key with its standard note name. The white keys (A through G) play the *natural* notes. The black keys play the

accidentals, the sharped and flatted notes. Black keys produce tones one-half note higher (sharp) or one-half note lower (flat) than the white keys beside them.

We now have all the necessary data to develop a music-playing program. What should this program do? It should look up a series of frequencies and durations in two tables and make repeated calls to SOUND to play the notes through the speaker.

The frequencies are, of course, obtained from Figure 8-2. The durations depend on what tempo you want and how many beats the note should be sustained. In 4/4 time, which has four beats to a measure, a *whole note* is

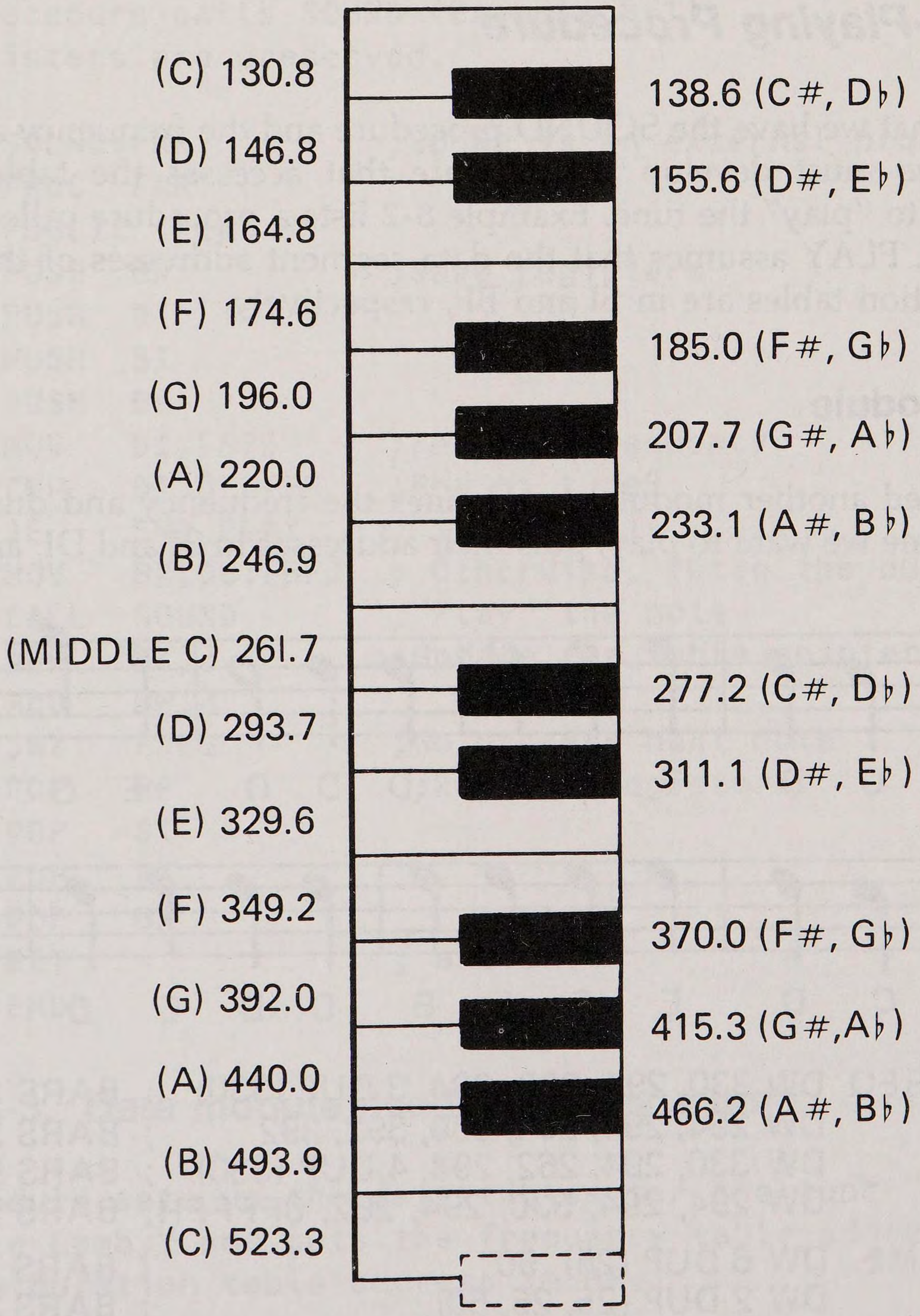


Figure 8-2. Two octaves on a piano keyboard.

sustained for four beats, a *half note* for two beats, a *quarter note* for one beat, an *eighth note* for one-half beat, and so on.

As a simple illustration, Figure 8-3 shows the music for "Mary Had a Little Lamb," along with its frequency and duration tables. Most are quarter notes, but the second and fourth bars end with half notes (E and G, respectively) and the final bar consists of a whole note (C).

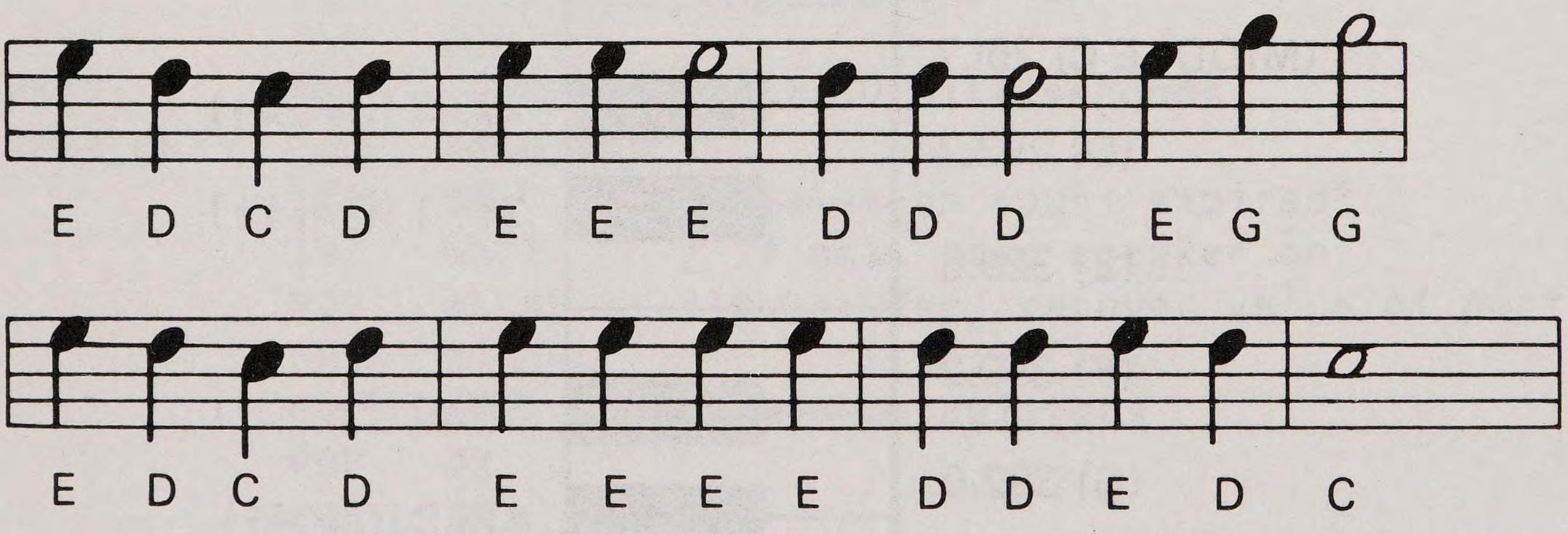
To build the duration table, we arbitrarily give a whole note a duration of one second. Therefore, a half note lasts a half-second and a quarter note lasts a quarter-second. The final value in the frequency table (0FFFFH) is not a note at all; it is a code that tells the note-playing procedure when to stop.

Music-Playing Procedure

Now that we have the SOUND procedure and the frequency and duration tables, we must develop a procedure that accesses the tables and calls SOUND to "play" the tune. Example 8-2 lists a procedure called PLAY that does this. PLAY assumes that the data segment addresses of the frequency and duration tables are in SI and BP, respectively.

Tune Module

We need another module that defines the frequency and duration tables for the tune we want to play, puts their addresses in SI and DI, and then calls




```
MARY FREQ  DW 330, 294, 262, 294, 3 DUP (330)   ; BARS 1 & 2
           DW 294, 294, 294, 330, 392, 392      ; BARS 3 & 4
           DW 330, 294, 262, 294, 4 DUP (330)   ; BARS 5 & 6
           DW 294, 294, 330, 294, 262, 0FFFFH;  BARS 7 & 8
MARY TIME  DW 6 DUP (25), 50                    ; BARS 1 & 2
           DW 2 DUP (25, 25, 50)                ; BARS 3 & 4
           DW 12 DUP (25), 100                  ; BARS 5 – 8
```

Figure 8-3. "Mary Had a Little Lamb."

PLAY. Example 8-3 shows this tune module for "Mary Had a Little Lamb." Just for fun, we've also included data tables for the lively old standard "Turkey In the Straw" (Example 8-4).

Example 8-2. Music-playing procedure.

```
;  This procedure "plays" a tune through the internal speaker,
;  based on two tables in the data segment.
;  Enter with the address of the frequency table in SI
;  and the address of the duration table in BP.
;  The frequency table must end with the value OFFFFH.
;  This procedure calls SOUND (Example 8-1).
;  All registers are preserved.
;
EXTRN     SOUND:FAR           ;SOUND is an external procedure
PLAY      PROC  FAR
          PUBLIC  PLAY
          PUSH  BX            ;Save registers
          PUSH  DI
          PUSH  SI
          PUSH  BP
FREQ:     MOV   DI,[SI]       ;Fetch a frequency
          CMP   DI,OFFFFH     ;End of tune?
          JE    END_PLAY      ; If so, exit
          MOV   BX,DS:[BP]    ; Otherwise, fetch the duration
          CALL  SOUND         ;"Play" the note
          ADD   SI,2          ;Update the table pointers
          ADD   BP,2
          JNZ   FREQ          ;Go process next note
END_PLAY: POP   BP             ;Restore registers
          POP   SI
          POP   DI
          POP   BX
          RET                 ; and exit
PLAY      ENDP
```

Example 8-3. Data module needed to play a tune.

```
;  This module sets up the data tables for the tune "Mary Had
;  a Little Lamb," and puts the frequency table address in SI
;  and the duration table address in BP.
;
;  This is the stack segment.
;
```

```
EXTRN       PLAY:FAR
MARY_STACK  SEGMENT  PARA STACK 'STACK'
            DB       64 DUP('STACK   ')
MARY_STACK  ENDS
;
;  This is the data segment.
;
MARY_DATA   SEGMENT  PARA 'DATA'
MARY_FREQ   DW  330,294,262,294,3 DUP(330)        ;Bars 1 & 2
            DW  294,294,294,330,392,392           ;Bars 3 & 4
            DW  330,294,262,294,4 DUP(330)        ;Bars 5 & 6
            DW  294,294,330,294,262,0FFFFH        ;Bars 7 & 8
;
MARY_TIME   DW  6 DUP(25),50                      ;Bars 1 & 2
            DW  2 DUP(25,25,50)                   ;Bars 3 & 4
            DW  12 DUP(25),100                    ;Bars 5-8
MARY_DATA   ENDS
;
;  This is the code segment.
;
MARY_CODE   SEGMENT  PARA 'CODE'
MARY_PROC   PROC     FAR
            ASSUME   CS:MARY_CODE,DS:MARY_DATA,SS:MARY_STACK
;
;  Set up the stack to contain the proper values so this
;  procedure can return to DOS.
;
            PUSH     DS           ;Put return seg. addr. on stack
            SUB      AX,AX        ;Clear a register
            PUSH     AX           ;Put zero return addr. on stack
;
;  Initialize the data segment address.
;
            MOV      AX,SEG MARY_DATA
            MOV      DS,AX
;
;  Initialize SI and BP, then call PLAY.
;
            LEA      SI,MARY_FREQ
            LEA      BP,DS:MARY_TIME
            CALL     PLAY
            RET
MARY_PROC   ENDP
MARY_CODE   ENDS
            END      MARY_PROC
```

Example 8-4. Data tables for "Turkey In the Straw."

```
TRKY_FREQ   DW   294,262,233,233,262,233,175,147,156   ;Line 1
            DW   175,196,175,147,175,233,262,294,294   ;Line 2
            DW   294,262,233,262
            DW   294,262,262,294,262,233,233,262       ;Line 3
            DW   233,175,147,156
            DW   175,196,175,147,175,233,262,294,349   ;Line 4
            DW   349,294,233,262
            DW   294,262,233,294,349,294,349,349       ;Line 5
            DW   294,349,294,349,311,392,311,392,392   ;Line 6
            DW   311,392,311,392,466,466,349,349       ;Line 7
            DW   294,294,262,233,262,294,349,392       ;Line 8
            DW   349,294,233,262,294,262,233,0FFFFH

TRKY_TIME   DW   2 DUP(12),25,6 DUP(12)                ;Line 1
            DW   4 DUP(12),25,2 DUP(12),2 DUP(25)      ;Line 2
            DW   4 DUP(12)
            DW   3 DUP(25),2 DUP(12),25,6 DUP(12)      ;Line 3
            DW   4 DUP(12),25,2 DUP(12),2 DUP(25)      ;Line 4
            DW   4 DUP(12)
            DW   2 DUP(25),50,12,25,12,2 DUP(25)       ;Line 5
            DW   12,25,12,50,2 DUP(12,25),25           ;Line 6
            DW   12,25,12,50,4 DUP(25)                 ;Line 7
            DW   3 DUP(25),3 DUP(12),25,5 DUP(12)      ;Line 8
            DW   3 DUP(25)
```

Music from the Keyboard

Since we already know which frequencies produce piano notes, we can go a step further and develop a procedure that makes the speaker sound a note when you press the corresponding letter key on the computer's keyboard. Although we probably cannot arrive at a reasonable approach that lets you produce all 88 notes that are on a piano, we can easily provide notes for the two octaves shown in Figure 8-2. To do this, we design the program so that you get lower-octave notes by pressing just a letter key (A through G) and upper-octave notes by pressing Shift and a letter key.

What about sharps and flats? The computer has no flat symbol, but it does have the sharp symbol (#) as the uppercase character on the 3 key. Therefore, we will let you press 3 to select a sharp and (arbitrarily) 2 to select a flat.

When composing, musicians put sharp or flat symbols immediately after the note name (e.g., they write 'A sharp' as *A#*). Thus, the ideal computer procedure is probably one in which the player produces a sharped note by

pressing a letter key, then 3 (for example, A3). However, that requires the computer to wait after a letter key to see whether the next key is 2 (sharp), 3 (flat), or another letter key. To avoid this, we ask the player to indicate sharp or flat *before* pressing the letter key.

Example 8-5 shows a procedure called KBD_MUSIC that lets you "play" piano notes from the keyboard. Note that is has the frequencies for two octaves in a data segment table and calls SOUND (Example 8-1) to play each note for 1/10 second.

While KBD_MUSIC is running, press a letter key to play a lower-octave note or Shift-letter key to play an upper-octave note. To produce a flatted or sharped note, press 2 or 3 (with Shift, if appropriate) before selecting the note. Press the Return key to exit to DOS.

Example 8-5. Music from the keyboard.

```
;  This procedure makes the computer keyboard function like a
;  two-octave piano keyboard (Figure 8-2).
;  To play a lower-octave natural note, press a letter key (A
;  through G).  To play an upper-octave natural note, hold Shift
;  down and press a letter key.  To play a sharp or flat, press
;  the 2 or 3 key, then the letter key.  Press Return to exit.
;  This procedure calls SOUND (Example 8-1).
;  All registers are preserved.
            PAGE      ,132
EXTRN       SOUND:FAR                 ;SOUND is an external procedure
KYBD_STACK      SEGMENT   PARA STACK 'STACK'
            DB    64 DUP('STACK   ')
KYBD_STACK      ENDS
KYBD_DATA       SEGMENT   PARA 'DATA'
FREQS       DW    220,247,131,147,165,175,196   ;Naturals, lower oct.
            DW    440,494,262,294,330,349,392   ;Naturals, upper oct.
            DW    233,262,139,156,175,185,208   ;Sharps, lower oct.
            DW    466,523,277,311,349,370,415   ;Sharps, upper oct.
            DW    208,233,123,136,156,165,185   ;Flats, lower oct.
            DW    415,466,247,277,311,330,370   ;Flats, upper oct.
KYBD_DATA       ENDS
;
;  This is the code segment.
;
KYBD_CODE       SEGMENT   PARA 'CODE'
KBD_MUSIC       PROC      FAR
                ASSUME    CS:KYBD_CODE,DS:KYBD_DATA,SS:KYBD_STACK
;
;  Set up the stack so this program can return to DOS.
```

```
;
                PUSH DS             ;Put return seg. addr. on stack
                MOV  CX,0           ;Clear a register
                PUSH CX
;
;  Initialize the data segment address.
;
                MOV  AX,SEG KYBD_DATA
                MOV  DS,AX
                PUSH AX             ;Save registers
                PUSH BX
                PUSH DI
                PUSH SI
NEW_NOTE:       MOV  AH,0           ;Read the next key
                INT  16H
                CMP  AL,0DH         ;Is it the Return key?
                JNE  LWR_FLT
                POP  SI             ; If so, restore registers
                POP  DI
                POP  BX
                POP  AX
                RET                 ;  and exit.
LWR_FLT:        SUB  SI,SI
                CMP  AL,'2'
                JNE  UPR_FLT
                MOV  BX,56          ;Possible lower-octave flat.
GET_LC:         MOV  AH,0
                INT  16H
                JMP  SHORT CHK_LO
UPR_FLT:        CMP  AL,'@'
                JNE  LWR_SHP
                MOV  BX,70          ;Possible upper-octave flat.
GET_UC:         MOV  AH,0
                INT  16H
                JMP  SHORT CHK_UO
LWR_SHP:        CMP  AL,'3'
                JNE  UPR_SHP
                MOV  BX,28          ;Possible lower-octave sharp.
                JMP  SHORT GET_LC
UPR_SHP:        CMP  AL,'#'
                JNE  LWR_NAT
                MOV  BX,42          ;Possible upper-octave sharp.
                JMP  SHORT GET_UC
LWR_NAT:        MOV  BX,0           ;Possible lower-octave natural.
                CMP  AL,'a'
```

```
                JB    UPR_NAT
                CMP   AL,'g'
                JA    NEW_NOTE
                JNA   NEW_FREQ
UPR_NAT:        MOV   BX,14       ;Possible upper-octave natural.
                JMP   SHORT CHK_UO
;
;  Check whether the key selects lower octave (a - g).
;
CHK_LO:         CMP   AL,'a'
                JB    NEW_NOTE
                CMP   AL,'g'
                JA    NEW_NOTE
                JNA   NEW_FREQ
;
;  Check whether the key selects upper octave (A - G).
;
CHK_UO:         CMP   AL,'A'
                JB    NEW_NOTE
                CMP   AL,'G'
                JA    NEW_NOTE
;
;  Convert the ASCII key code into an index in SI, then use it to
;  look up a frequency and call SOUND to play the note 0.1 second.
;
NEW_FREQ:       AND   AL,0FH
                SHL   AL,1
                SUB   AH,AH
                ADD   SI,AX
                MOV   DI,FREQS-2[BX][SI]
                MOV   BX,10             ;Duration is 1/10 second.
                CALL  SOUND
                JMP   SHORT NEW_NOTE
KBD_MUSIC       ENDP
KYBD_CODE       ENDS
                END   KBD_MUSIC
```

9

Macros

Macros are subroutine-like "miniprograms" you can insert in source programs by mentioning their names. In this chapter we will describe how to create macros and use them in programs. Then we will present a set of useful macros and show you how to put them into a "macro library." The macro facility is part of IBM's Macro Assembler (MASM); you cannot use it with the Small Assembler.

9.1 Introduction to Macros

As we just mentioned, a macro is a sequence of assembler statements (instructions and pseudo-ops) that may appear several times in a program. Like procedures, macros have names. Once a macro has been defined, you can enter its name in your source program wherever you would normally type the instruction sequence.

Macros vs Procedures

Although macros and procedures both provide a shorthand reference to a frequently-used instruction sequence, they are not the same. The code for a procedure occurs once in a program, and the processor transfers to it (i.e., *CALLs* it) as needed. By contrast, the code for a macro may occur many times within a program; the assembler replaces each occurrence of a macro name with the instructions that name represents. (We say the assembler "expands" the macro.) Therefore, when you execute the program, the processor executes the macro instructions "in-line," without transferring elsewhere in memory, as it does with a procedure. Hence, *a macro name is a user-defined assembler directive;* it provides commands to the assembler, rather than to the microprocessor.

Macros have three advantages over procedures:

1. Macros are *dynamic.* You can easily change how the macro operates (not merely what it operates on) each time, by changing its input parameters. By contrast, you can only vary the data that gets passed to a procedure, making procedures much more inflexible.
2. Macros make faster-executing programs, because the processor is not delayed by call and return instructions, as it is with procedures.
3. Macros can be entered into a *macro library* that programmers can draw from to create other programs.

Nothing comes for free, however. Macros have a major disadvantage that procedures do not have: since a macro gets expanded every time it is used, it tends to make machine language programs longer by filling memory with repeated instruction sequences.

Macros Speed Up Programming

Macros can speed up your programming and debugging work, as well as any program updating you might do in the future. They speed up your programming work because you can create a macro just once, then use it wherever it is needed in a program. Rather than entering a certain set of instructions, you enter only the macro name that represents them. (Macros have this in common with subroutines.)

Macros can speed up your debugging because you create and debug them individually. Once a macro is working properly, you should never have to worry about whether *that* portion of your program is correct. You can concentrate on finding errors in other portions.

Programs that include macros are generally easier to read and understand. That means they are also easier to update. For example, suppose you want to clear the screen, then move the cursor to the upper left-hand corner. To do this, enter the following instruction sequences:

```
;
;  These instructions clear the screen.
;
        MOV   CH,0     ;Start at row 0
        MOV   CL,0     ; and column 0.
        MOV   DH,24    ;End at row 24
        MOV   DL,79    ; and column 79.
        MOV   AH,6     ;Select scroll-up option.
        MOV   AL,0     ;Clear entire screen.
        MOV   BH,7
        INT   10H      ;Issue Video I/O interrupt.
;
;  These instructions move the cursor to the "home" position
```

```
;
        MOV   DH,0     ;Select row 0
        MOV   DL,0     ; and column 0.
        MOV   AH,2     ;Select cursor-moving option.
        INT   10H      ;Move the cursor.
```

However, if you define these instruction sequences as macros, enter the following instead:

```
        CLS    ;Clear the screen.
        HOME   ;Move cursor to home position.
```

Which program would you expect someone else to be able to understand more readily? For that matter, which would *you* want to tackle, say, six months later?

Finally, macros also make a program easier to change. Change the macro definition and the assembler automatically uses the new version everywhere it used the old one previously.

Contents of Macros

Every macro definition has three parts:

1. *Header*—The MACRO pseudo-op, with the *name* of the macro in the label field and, optionally, a *dummy-list* in the operand field. The *dummy-list* identifies variables—input parameters you can change each time you call the macro.
2. *Body*—The sequence of assembler statements (instructions and pseudo-ops) that define what the macro does.
3. *Terminator*—The ENDM pseudo-op, which marks the end of the macro definition. If you omit ENDM, the assembler will display the error message "End of file encountered on input file."

For example, the following is a simple macro to add word-size values:

```
ADD_WORDS   MACRO   TERM1,TERM2,SUM
        MOV   AX,TERM1
        ADD   AX,TERM2
        MOV   SUM,AX
              ENDM
```

The assembler doesn't care whether you specify register names, memory locations, or immediate values for the operands (you can't use an immediate for SUM, of course). As long as the final form is legal, the assembler will

make the substitutions without complaining. For example, at one place in a program, you could add two memory locations by entering

```
ADD_WORDS  PRICE,TAX,COST
```

This makes the assembler insert the following instructions in the program:

```
MOV  AX,PRICE
ADD  AX,TAX
MOV  COST,AX
```

Somewhere else you can add two registers by entering

```
ADD_WORDS  BX,CX,DX
```

This time the assembler inserts

```
MOV  AX,BX
ADD  AX,CX
MOV  DX,AX
```

Note how much easier it is to pass parameters to macros than to procedures. With a macro, you just enter the parameter; with a procedure, you must put in a register or memory location.

9.2 Macro Pseudo-Ops

Table 9-1 summarizes the macro pseudo-ops provided by the IBM MACRO Assembler. We have divided them into four groups: general-purpose, repeat, conditional, and listing.

General-Purpose Pseudo-Ops

We have already discussed the MACRO pseudo-op; it gives the macro's name and lists the names of the dummy operands the macro uses, if any.

LOCAL Pseudo-Op

If your macro contains labeled instructions or pseudo-ops, you must tell the assembler to change the labels every time it expands the macro. Otherwise you end up with "Symbol is Multi-Defined" errors. The LOCAL pseudo-op tells the assembler which labels to change each time.

For example, the following macro (WAIT) makes the processor wait until the value COUNT has been decremented to zero. Declaring the label *NEXT* LOCAL allows us to use this macro more than once in a program.

```
WAIT  MACRO  COUNT
       LOCAL  NEXT
       PUSH   CX              ;Save current CX.
       MOV    CX,COUNT        ;Move count to CX.
NEXT:  LOOP   NEXT            ;Loop until count is 0.
       POP    CX              ;Restore CX.
      ENDM
```

Table 9-1. Macro pseudo-ops.

Pseudo-Op	*Function*

General-Purpose

MACRO Format: `name MACRO [dummy-list]`
`..`
`..`
`ENDM`

Assigns a *name* to a sequence of assembler statements. Every MACRO definition must end with an ENDM pseudo-op.

LOCAL Format: `LOCAL dummy-list`

Makes the assembler create a unique symbol for each entry in *dummy-list,* and substitute that symbol for each occurrence of the entry in the expansion.

Repeat

IRP Format: `IRP dummy,<argument-list>`
`..`
`..`
`ENDM`

Makes the assembler repeat the statements between IRP and ENDM once for each argument in <*argument-list*>. Each repetition substitutes the next item in <*argument-list*> for each occurrence of *dummy* in the block.

IRPC Format: `IRPC dummy,string`
`..`
`..`
`ENDM`

Makes the assembler repeat the statements between IRPC and ENDM once for each character in *string.* Each repetition substitutes the next character in *string* for every occurrence of *dummy* in the block.

Table 9-1. Macro pseudo-ops (continued).

Pseudo-Op	*Function*

REPT	Format: `REPT expression`
	`..`
	`..`
	`ENDM`
	Makes the assembler repeat the statements between REPT and ENDM *expression* times.

Conditional	
EXITM	Format: `EXITM` Terminates a macro expansion based on the result of a conditional pseudo-op.

IF1	Format: `IF1 expression`
	`..`
	`..`
	`ENDIF`
	True if assembler is executing pass 1. Generally used to INCLUDE a macro library file in a source program.

IFB	Format: `IFB <argument>`
	`..`
	`..`
	`ENDIF`
	True if <*argument*> is blank. The angle brackets are required.

IFNB	Format: `IFNB <argument>`
	`..`
	`..`
	`ENDIF`
	True if <*argument*> is not blank. The angle brackets are required.

Listing	
.LALL	Format: `.LALL` Lists the complete macro text (including comments) for all expansions.
.SALL	Format: `.SALL` Excludes macro text from listings.
.XALL	Format: `.XALL` Lists only macro lines that generate object code. This is the default setting.

Note that LOCAL immediately follows the MACRO statement. If used, *LOCALs must be the first statements in a macro.* They must precede everything else, even comments.

Declaring a label LOCAL also means you can reuse it in other macros. The assembler gives it a new internal name each time it expands the macro; hence, no duplication.

Repeat Pseudo-Ops

The REPT, IRP, and IRPC pseudo-ops make the assembler repeat sequences of assembler statements in a macro.

REPT obtains its repetition count from an *expression* in the operand field. For example, the following macro allocates *LENGTH* bytes in memory, and initializes them with the values 1 through *LENGTH*, respectively:

```
ALLOCATE  MACRO  TLABEL,LENGTH
TLABEL  EQU  THIS BYTE
VALUE  =  0
  REPT  LENGTH
  VALUE  =   VALUE+1
         DB  VALUE
  ENDM
          ENDM
```

Note that we need two ENDMs here; the first marks the end of the REPT, the second marks the end of the MACRO definition.

After defining ALLOCATE, we could use it to set up a 40-byte table, called TABLE1, with this sequence:

```
DATA    SEGMENT  PARA 'DATA'
ALLOCATE  TABLE1,40
DATA    ENDS
```

The second repeating pseudo-op, IRP, lets you list the arguments that are to be substituted for a *dummy* symbol with each repetition. For example, the sequence

```
IRP  VALUE,<1,2,3,5,7,11,13,17,19,23>
  DW  VALUE*VALUE*VALUE
ENDM
```

sets up a 10-word table that contains the cubes for the first 10 prime numbers.

IRPC is the same as IRP, but its arguments are string variables rather than numbers. IRPC takes two operands, a *dummy* symbol and a *string*, and repeats the statements in the block once for each character in the string. Each repeti-

tion substitutes the next character in the string for every occurrence of *dummy* in the block.

For example, the sequence

```
IRPC   CHAR,0123456789
  DB   CHAR
ENDM
```

sets up a 10-byte text string in memory that contains the ASCII codes for the digits 0 through 9.

Conditional Pseudo-Ops

These pseudo-ops are similar to the conditionals we discussed in Section 2.8. That is, the assembler tests whether the condition is satisfied. If so, it assembles the statements between IF and ENDIF; if not, it skips them.

IF1 Pseudo-Op

The IF1 (If Pass 1) pseudo-op is used to read a macro library file into a source program. We will discuss it further in Section 9.4.

IFB Pseudo-Op

If you enter fewer parameters than a macro expects, the assembler will set the omitted or *blank* parameters to zero unless you tell it otherwise. IFB (If Blank) lets you specify some alternate course of action for blank parameters. It is commonly used to make a macro terminate early if some needed parameter is missing. We will talk more about terminating early when we discuss EXITM later in this section.

IFNB Pseudo-Op

When the assembler encounters IFNB (If Not Blank), it assembles the enclosed instructions only if the user has supplied a value for the parameter; otherwise, it skips them.

For example, a macro that reads a name may expect you to call it using the form

```
GET_NAME   FIRST-NAME,MIDDLE-INITIAL,LAST-NAME
```

The macro definition for GET_NAME would include instructions that obtain the first name, then the middle initial, then the last name. However, since not all people have middle initials, you must make some provision for omitting them. You could do this by using IFNB to include the middle-initial instructions only if the user supplies an initial. Hence, GET_NAME's definition would have the following general form:

```
GET_NAME  MACRO  first-name,middle-initial,last-name
          ..     (These instructions read the first name)
          ..
IFNB <middle-initial>
          ..     (These instructions read the middle initial)
          ..
ENDIF
          ..     (These instructions read the last name)
          ..
          ENDM
```

With this safeguard, you may now use a form like *GET_NAME John,,Brown* without wondering how the assembler will react to the missing initial.

IFNB can also help you avoid assembly errors resulting from missing operands. For example, if your macro includes the statement *PUSH reg_name*, and you omit *reg_name* from the parameter list, the assembler will try to assemble it as *PUSH 0*—which is, of course, illegal. To protect against this, use

```
IFNB  <reg_name>
  PUSH  reg_name
ENDIF
```

(In this case, you probably need similar protection for the matching *POP reg_name*.)

EXITM Pseudo-Op

The EXITM (Exit Macro) pseudo-op makes the assembler stop expanding a macro early, based on the result of a conditional pseudo-op, as in

```
IFB <name>
  EXITM
ENDIF
```

For example, recall the ALLOCATE macro that we defined with the REPT pseudo-op. The following redefines this macro so that the assembler allocates the table space only if the LENGTH parameter is less than 50:

```
ALL_LT_50  MACRO  LENGTH
      VALUE = 0
      IF LENGTH GE 50
        EXITM
      ENDIF
      REPT LENGTH
        VALUE = VALUE + 1
        DB  VALUE
      ENDM
           ENDM
```

Listing Pseudo-Ops

The .LALL, .SALL, and .XALL pseudo-ops let you control how much macro text gets included in an assembly listing (LST). If you don't specify any of these options, the assembler assumes you want .XALL. That is, the listing will show only the lines that generate object code; it will omit stand-alone comments and pseudo-ops that don't reserve memory locations.

The .LALL pseudo-op produces a full listing, including comments, whereas .SALL omits all macro text from the listing. You might use .LALL to generate a copy of a program for your files, then reassemble using .SALL to generate the listing you use once the macros have been completely debugged.

9.3 Macro Operators

The MACRO Assembler also provides four operators that you can use with macros; see Table 9-2.

& Operator

The & operator lets you create custom labels and operands. For example, the following macro sets up a byte table that has a specified name and length:

```
DEF_TABLE   MACRO   SUFFIX,LENGTH
      TABLE&SUFFIX   DB   LENGTH DUP(?)
             ENDM
```

Then, if you enter *DEF_TABLE A,5* in a program, the assembler converts it to

```
TABLEA   DB   5 DUP(?)
```

Table 9-2. Macro operators.

Operator	*Function*
&	Format: `text&text` Concatenates (joins) text or symbols.
;;	Format: `;;comment` Omits *comment* from macro expansion and from listing, even one produced with .LALL.
!	Format: `!character` Used in an argument to make the assembler use the *character* as a literal value, rather than as a symbol.
%	Format: `%symbol` Converts *symbol* to a number. When the assembler expands the macro, it substitutes the number for the symbol.

;; Operator

The ;; operator makes the assembler omit comments from the macro expansion. Without comments, your final program will take up less memory and will therefore assemble faster. When defining macros, use the regular ; operator only for the comments that are absolutely necessary, use ;; for the rest.

9.4 Defining Macros in Source Programs

There are two ways to use macros: you can enter their definitions at the beginning of the program or read them into the program from a separate macro library file. In this section we describe how to enter macros directly into a program. We will discuss macro libraries in Section 9.5.

If you have a special-purpose macro that you need in only one program, you can define it in the program, then call it as needed. Let's look at an illustration.

Example 7-1 in the graphics chapter lists a program that draws a diagonal line of "smile face" characters down the screen. Example 9-1 shows a new version that uses macros to transfer registers to and from the stack (PUSH_REGS and POP_REGS), move the cursor (MOVE_CURSOR), and display the face character (PRINT_AL). Although the total text is longer—

because of the macro definitions—the actual procedure text is quite a bit shorter, and more readable.

The disadvantage of entering macros directly into a program is that they apply only to that program. To use them in other programs, you can put them in a macro library.

Example 9-1. Diagonal line program using macros.

```
TITLE  Diagonal Line Program with Macros (NEW_DIAG.ASM)
;*************************************************
;  Macro definitions.
;*************************************************
PUSH_REGS  MACRO  reg_list
;;
;;  Save registers on the stack.
;;
    IRP  reg,<reg_list>
         PUSH  reg
    ENDM
           ENDM
POP_REGS  MACRO  reg_list
;;
;;  Retrieve registers from the stack.
;;
    IRP  reg,<reg_list>
         POP   reg
    ENDM
          ENDM
MOVE_CURSOR  MACRO
;;
;; Move the cursor to the row and column positions in DH and DL,
;;  respectively.
;;
         MOV   AH,2    ;;Select cursor-moving option.
         INT   10H
             ENDM
PRINT_AL  MACRO
;;
;;  Display the character in AL.
;;
         MOV   CX,1    ;;Display just one character.
         MOV   AH,14   ;;Select display option.
         INT   10H
           ENDM
```

```
;*****************************************************************
;  Beginning of program.
;*****************************************************************
STACK   SEGMENT  PARA STACK 'STACK'
        DB  64 DUP('STACK   ')
STACK   ENDS
OUR_CODE  SEGMENT  PARA 'CODE'
DIAG_LINE  PROC  FAR
           ASSUME  CS:OUR_CODE,SS:STACK
;
;  Set up the stack to contain the proper values so this program
;  can return to DOS.
;
        PUSH  DS        ;Put return seg. addr. on stack.
        MOV   DI,0      ;Clear a register.
        PUSH  DI        ;Put zero return addr. on stack.
;
;  This is the main procedure.
;
        PUSH_REGS  <AX,BX,CX,DX>  ;Save registers.
        STI               ;Enable interrupts.
        MOV   AH,15     ;Set BH to active display page.
        INT   10H
        MOV   AH,0      ;Select 80x25, B/W, Alpha.
        INT   10H
        MOV   DX,0      ;Start at row 0, column 0.
SET_CRSR:  MOVE_CURSOR  ;Set new cursor position.
        MOV   AL,2      ;Display char. is smile face.
        PRINT_AL
        INC   DH        ;Point to next row
        INC   DL        ; and new column.
        CMP   DH,25     ;Bottom of screen?
        JNE   SET_CRSR
        POP_REGS  <DX,CX,BX,AX>    ;Yes.  Restore registers
        RET                        ; and exit.
DIAG_LINE  ENDP
OUR_CODE   ENDS
           END   DIAG_LINE
```

9.5 Macro Library

A macro library is a disk file that contains definitions of macros you may need in several programs. Once you have created this file, you can read it

into any source program. This makes all of the macros in the library available to that program. To use any one of them, just mention its name.

Creating a Macro Library

A macro library is simply a source file that includes macro definitions. Therefore, you can create the library file using EDLIN or a word processor, just as you create a standard program.

Guidelines for Defining Macros

The macros in a macro library should be general-purpose routines that you can use in virtually any program. Therefore, you should design them so that they do their intended tasks, but ensure that they won't conflict with any program that uses them. Here are a few guidelines that should help you to design efficient macros:

1. Document the macros as thoroughly as possible. Include lots of comments. Remember, your macro definitions should be meaningful to anyone who has to use them, not just to you.
2. Use the ;; operator to enter comments and use the Tab key (rather than the space bar) to separate fields. These steps help keep the size of your programs to a minimum, which means that they will assemble faster.
3. Keep macros as general as possible. If you need a special-purpose macro, define it using a more general macro, if possible. For example, the macro library we will introduce in this chapter includes a macro called LOCATE that moves the cursor to a specified row and column position. Then we define a separate macro called HOME that moves the cursor to the upper left-hand corner of the screen. HOME's definition is simply LOCATE 0,0.
4. If the macro includes labels, list them in a LOCAL statement.
5. Save every register that the macro uses except output registers. Do this as usual, with PUSHes at the beginning of the macro and POPs at the end.
6. If a macro definition requires a task that is performed by a previously-defined macro, call that macro to do it.

Reading a Macro Library Into a Program

To read a macro library into a source program, you must give the assembler its name with an INCLUDE statement. However, if you do this with

only, say, INCLUDE MACRO.LIB, the assembler unnecessarily reads the library during both pass 1 and pass 2.

To avoid this repetition, you should put the INCLUDE in an IF1 *conditional structure,* as follows:

```
IF1
   INCLUDE MACRO.LIB
ENDIF
```

Doing this makes the assembler read the library during pass 1, but ignore it during pass 2. Moreover, since the assembler produces the listing during pass 2, the INCLUDEd text will not appear on it.

Purging Macros

A disadvantage of using a macro library is that when the assembler INCLUDEs it, all of its macros get read in and stored in the assembler's work space in memory. This includes macros that are not going to be used in this assembly, thus wasting work space and possibly creating an "out of memory" error condition. To avoid this error, you can *purge* unneeded macros and thereby free up the space occupied by their definitions. To do this, simply list the purge victims immediately after your INCLUDE, as in

```
INCLUDE MACRO.LIB
PURGE MAC1, MAC2, MAC3
```

Macro Descriptions

In the remainder of this chapter we will describe and define more than 30 macros that you might want to put in a macro library. Included are macros that perform common DOS and BASIC functions (e.g., clear the screen, read keyboard data) and others that perform common assembly language tasks (e.g., save registers, convert between strings and numbers).

Table 9-3 lists the names of the macros, divided into seven functional groups: general-purpose, conversion, cursor-related, display, keyboard, sound, and time. The subsections that follow present more detailed information.

Table 9-3. Macros in the macro library.

General-Purpose Macros

DELAY	POP_REGS	SET_ES
MAKE_STACK	RAND	
PUSH_REGS	SET_DS	

Table 9-3. Macros in the macro library (continued).

Conversion Macros		
$2BIN	BIN2$	KEY_$2BIN
Cursor-Related Macros		
CLS	HOME	MOVE_CURSOR
CR	LF	POS
CRLF	LOCATE	TAB
Display Macros		
MESSAGE	PRINT_NUMBER	SHOW_AL
MESSAGE_DX	PRINT$	VIDEO_STATE
PRINT_AL	PRINT_DX	
Keyboard Macros		
INKEY	IN$	
INKEY_I	IN$_DX	
Sound Macros		
BEEP	SOUND	SOUND_DI_BX
Time Macros		
READ_TIME	SET_TIME	

General-Purpose Macros

DELAY [,minutes][,seconds][,hundredths]

Action: Makes the computer wait for a specified interval.

Inputs: Optional **minutes** (0 to 59), **seconds** (0 to 59), and **hundredths** of a second (0 to 99).

Outputs: None.

Examples:

```
DELAY ,30,50    ;Wait 30.5 seconds
DELAY 1,30      ;Wait 1.5 minutes
```

MAKE_STACK stack-name

Action: Creates a standard stack segment (that is, reserves 64 repetitions of 'STACK ') and gives it the specified name.

Inputs: **stack-name** is name of stack segment.

Outputs: None.

Example:

```
MAKE_STACK MY_STACK   ;Stack seg. is named MY_STACK
```

PUSH_REGS <reg0[,reg1,...]>

Action: Pushes registers on the stack, in the specified order.
Inputs: Register names, separated by commas and enclosed with angle brackets.
Outputs: None.
Example:
```
PUSH_REGS <AX,BX,DX>    ;Save AX, BX, and DX
```

POP_REGS <reg0[,reg1,...]>

Action: Pops registers off the stack, in the specified order.
Inputs: Register names, separated by commas and enclosed with angle brackets.
Outputs: Specified registers change according to stack values.
Comments: List registers in the opposite order of PUSH_REGS.
Example:
```
POP_REGS <DX,BX,AX>    ;Retrieve DX, BX, and AX
```

RAND limit

Action: Produces a random integer between 0 and **limit.**
Inputs: **limit** (4 to 127)
Outputs: (AL) = Random number
Example:
```
RAND 85    ;Generate random number between 0 and 85
```

SET_DS dseg-name

Action: Loads the address of the data segment into DS.
Inputs: **dseg-name** is the name of a data segment.
Outputs: DS points to the data segment.

SET_ES eseg-name

Action: Loads the address of the extra segment into ES.
Inputs: **eseg-name** is name of extra segment.
Outputs: ES points to extra segment.

Conversion Macros

$2BIN string-name

Action: Converts a string in the data segment to a signed number in AX.
Inputs: **string-name** refers to the starting address of a string.
String Format: The string may be up to seven bytes long, where the first byte holds the character count. The first significant character may be a minus (-) or plus (+) sign, the rest must be numeric digits (0 through 9). For example,
```
MY_STRING   DB   6,'-16540'
```
Outputs: If the string is legal, CF = 0 and (AX) = Number
If the string is illegal, CF = 1

Comments: To convert a number entered from the keyboard, use KEY_$2BIN.

Example:
```
$2BIN  MY_STRING    ;Convert MY_STRING to number
JC     PRINT_ERROR
```

BIN2$ string-name

Action: Converts the signed contents of AX to a string in the data segment.

Inputs: **string-name** refers to the starting address of a target string.

String Format: Seven reserved bytes. For example,
```
MY_STRING  DB  7 DUP(?)
```

Output: None. AX is unaffected.

Example:
```
BIN2$  MY_STRING    ;Convert AX to string
```

KEY_$2BIN

Action: Converts a sequence of keyboard characters to a signed number in AX.

Inputs: None.

Outputs: If the string is legal, CF = 0 and (AX) = Number
If the string is illegal, CF = 1

Comments: To convert a string in the data segment, see $2BIN.

Example:
```
KEY_$2BIN              ;Read number from keyboard
JC  PRINT_ERROR
```

PRINT_NUMBER See *Display Macros.*

Cursor-Related Macros

CLS

Action: Clears the screen.

Inputs: None.

Outputs: None.

CR

Action: Carriage Return. Moves the cursor to the beginning of the current line.

Inputs: None.

Outputs: None.

Comments: See also CRLF.

CRLF

Action: Carriage Return/Line Feed. Moves the cursor to the beginning of the next line.

Inputs: None.

Outputs: None.

HOME

Action: Moves the cursor to the upper left-hand corner of the screen.
Inputs: None.
Outputs: None.
Comments: Same as LOCATE 0,0.

LF

Action: Line Feed. Moves the cursor to the next line.
Inputs: None.
Outputs: None.
Comments: (1) See also CRLF.
(2) If the cursor is already on the bottom line, LF moves it to the top line.

LOCATE row,column

Action: Moves cursor to the specified position.
Inputs: **row** (0 to 24) and **column** (0 to 79).
Outputs: None.
Comments: (1) Same as MOVE_CURSOR, but LOCATE works with user values rather than register values.
(2) If **row** exceeds 24, LOCATE moves cursor to line 0.
(3) If **column** exceeds 79, LOCATE moves cursor to column 0.
(4) See also HOME.
Example:

```
LOCATE 24,79  ;Move cursor to lower right corner.
```

MOVE_CURSOR

Action: Moves cursor to row in DH and column in DL.
Inputs: (DH) = Row number
(DL) = Column number
Outputs: None. DH and DL are unaffected.
Comments: (1) Similar to LOCATE, but MOVE_CURSOR works with register values rather than user values.
(2) If DH exceeds 24, cursor moves to line 0.
(3) If DL exceeds 79, cursor moves to column 0.
(4) Use POS to read current cursor position.

POS

Action: Reads current cursor position.
Inputs: None.
Outputs: (DH,DL) = Row, column numbers
(CH,CL) = Cursor mode

TAB column

Action: Moves the cursor to the specified column.
Inputs: **column** (0 to 79)

Outputs: None.
Comments: If the cursor is beyond the specified position, TAB moves it to that column on the next line.
Example:
```
TAB  40   ;Tab to column 40
```

Display Macros

MESSAGE string-name

Action: Displays a message string in the data segment.
Inputs: **string-name**
String Format: String must end with a $ character. For example,
```
ILLEGAL  DB  'Illegal input value.  Try again.$'
```
Outputs: None.
Comments: Same as MESSAGE_DX, but MESSAGE takes its string specifier from a user name rather than from a register.
Example:
```
MESSAGE  ILLEGAL   ;Display ILLEGAL message
```

MESSAGE_DX

Action: Displays a message string in the data segment whose offset is in DX.
Inputs: (DS:DX) = Starting address of a string.
String Format: Same as for MESSAGE.
Outputs: None. DS and DX are unaffected.
Comments: Same as MESSAGE, but MESSAGE_DX takes its string specifier from a register rather than from a user name.
Example:
```
LEA  DX,ILLEGAL   ;Display ILLEGAL message
MESSAGE_DX
```

PRINT_AL

Action: Displays the character in AL, then advances the cursor.
Inputs: (AL) = Character code
Outputs: None. AL is unaffected.
Comments: To keep the cursor at the display position, use SHOW_AL instead.
Example:
```
MOV  AL,'P'   ;Display "P", then move the cursor
PRINT_AL
```

PRINT_NUMBER

Action: Displays the signed number in AX.
Inputs: None.
Outputs: None. AX is unaffected.

PRINT$ string-name

Action: Displays a string in the data segment.
Inputs: **string-name**
String Format: String beginning with a character count byte.

Outputs: None.
Comments: Same as PRINT$_DX, but PRINT$ obtains its string specifier from a user name rather than from a register.

PRINT$_DX

Action: Displays a string in the data segment whose offset is in DX.
Inputs: (DS:DX) = Starting address of string.
String Format: Same as PRINT$.
Outputs: None. DS and DX are unaffected.
Comments: Same as PRINT$, but PRINT$_DX obtains its string specifier from a register rather than from a user name.

SHOW_AL

Action: Displays the character in AL, but does not move the cursor.
Inputs: (AL) = Character code
Outputs: None. AL is unaffected.
Comments: To advance the cursor after displaying the character, use PRINT_AL instead.
Example:

```
MOV  AL,'P'   ;Display "P" at the cursor position
SHOW_AL
```

VIDEO_STATE

Action: Reports video state.
Inputs: None.
Outputs: (AL) = Current mode
(AH) = Number of columns on screen
(BH) = Active page

Keyboard Macros

INKEY

Action: Waits for a key, then reads its ASCII code into AL and displays it.
Inputs: None.
Outputs: (AL) = ASCII code for key
Comments: (1) To read a key without displaying it, use INKEY_I instead.
(2) See Appendix B for ASCII codes.

INKEY_I

Action: Waits for a key, then reads its ASCII code into AL, but does not display it.
Inputs: None.
Outputs: (AL) = ASCII code for key
Comments: (1) To display the key you read, use INKEY instead.
(2) See Appendix B for ASCII codes.

IN$ string-name

Action: Reads a sequence of keyboard characters into a buffer in the data segment.

Inputs: **string-name** is name of buffer.

Buffer Format: The buffer must be three bytes longer than the string it is to hold. The first byte specifies the maximum string length plus one. For example, the following buffer accepts strings of up to 10 characters:

```
STRING_BUFF  DB  11,12 DUP(?)
```

Outputs: Second byte of the buffer contains the actual length of the input string, in bytes.

Comments: (1) Same as IN$_DX, but IN$ obtains the buffer specifier from the user rather than from a register.
(2) IN$ accepts keys until the user presses Enter. The computer beeps when the buffer is full.

Example:

```
IN$  STRING_BUFF    ;Read keys into buffer
```

IN$_DX

Action: Reads a sequence of keyboard characters into a buffer in the data segment.

Inputs: (DS:DX) = Starting address of the buffer

Buffer Format: Same as IN$.

Outputs: Second byte of the buffer contains the actual length of the input string, in bytes. DS and DX are unaffected.

Comments: (1) Same as IN$, but IN$_DX obtains the buffer specifier from a register rather than from the user.
(2) IN$_DX accepts keys until the user presses Enter. The computer beeps when the buffer is full.

Example:

```
LEA  DX,STRING_BUFF    ;Read keys into buffer
IN$_DX
```

KEY_$2BIN See *Conversion Macros.*

Sound Macros

BEEP

Action: Beeps the speaker.

Inputs: None.

Outputs: None.

Comments: Same as SOUND 1000,50.

SOUND freq,duration

Action: Produces a tone of a specified frequency and duration.

Inputs: **freq** in Hertz (21 to 65535) and **duration** in hundredths of a second (0 to 65535).

Outputs: None.
Comments: Same as SOUND_DI_BX, but SOUND bases the tone on user values rather than register values.
Example:
```
SOUND 262,100    ;Sound Middle C for one second
```

SOUND_DI_BX

Action: Produces a tone having the frequency specified in DI and the duration specified in BX.
Inputs: (DI) = Frequency in Hertz (21 to 65535).
(BX) = Duration in hundredths of a second (0 to 65535).
Outputs: None. DI and BX are unaffected.
Comments: Same as SOUND, but SOUND_DI_BX uses register inputs rather than user values.
Example:
```
MOV DI,262 ;Sound Middle C for one second
MOV BX,100
SOUND_DI_BX
```

Time Macros

READ_TIME

Action: Reads the current time.
Inputs: None.
Outputs: (CH) = Hours
(CL) = Minutes
(DH) = Seconds
(DL) = 1/100 seconds

SET_TIME [hours][,minutes][,seconds][,hundredths]

Action: Sets the time.
Inputs: Optional **hours** (0 to 23), **minutes** (0 to 59), **seconds** (0 to 59), and **hundredths** of a second (0 to 99). Omitted operands become 0.
Outputs: None.
Example:
```
SET_TIME 1,,30    ;Set time to 1:00:30 AM
```

Macro Definitions

The remainder of this chapter lists the definitions for each of the macros in the library, in alphabetical order. If you have suggestions about improving any of them, or want to suggest additional definitions, please contact the author through Brady Communications Company, Inc.; Bowie, MD 20715.

Cross-References

Many of the macros in the library call other macros. Therefore, if you change or delete a macro, it may affect others. Table 9-4 is a cross-reference table for the library. It lists which macros each macro definition uses and which ones it is used in.

Table 9-4. Cross-reference table for macro library.

Macro	*Used in*	*Uses*
$2BIN	KEY_$2BIN	–
BEEP	–	SOUND
BIN2$	PRINT_NUMBER	–
CLS	–	–
CR	CRLF	MOVE_CURSOR, POS
CRLF	–	CR, LF
DELAY	–	READ_TIME
HOME	–	LOCATE
IN$	KEY_$2BIN	–
IN$_DX	IN$	–
INKEY	–	–
INKEY_I	–	–
KEY_$2BIN	–	$2BIN, IN$
LF	CRLF, TAB	MOVE_CURSOR, POS
LOCATE	HOME	MOVE_CURSOR
MAKE_STACK	–	–
MESSAGE	–	MESSAGE_DX
MESSAGE_DX	MESSAGE	–
MOVE_CURSOR	CR, LF, LOCATE	VIDEO_STATE
POP_REGS	–	–
POS	CR, LF, TAB	VIDEO_STATE
PRINT_AL	PRINT$_DX	VIDEO_STATE
PRINT_NUMBER	–	BIN2$, PRINT$
PRINT$	PRINT_NUMBER	PRINT$_DX
PRINT$_DX	PRINT$	–
PUSH_REGS	–	–
RAND	–	–
READ_TIME	DELAY	–
SET_DS	–	–
SET_ES	–	–
SET_TIME	–	–
SHOW_AL	–	VIDEO_STATE
SOUND	BEEP	SOUND_DI_BX
SOUND_DI_BX	SOUND	–
TAB	–	LF, POS, VIDEO_STATE
VIDEO_STATE	CR, LF, MOVE_CURSOR, POS, PRINT_AL, SHOW_AL, TAB	–

$2BIN MACRO string_name

```
        LOCAL   BLANKS,CHK_NEG,CHK_POS,GO_CONV,GOOD,NO_GOOD,THRU
        LOCAL   CONV_AB,RANGE,NON_DIG,DIGIT,END_CONV,SKIPIT
;;
;;  Convert the specified string to a signed binary number
;;  in AX.  The first byte in the string must hold the
;;  character count.
;;  If the conversion is successful, CF = 0;; otherwise CF = 1.
;;
        PUSH    BX                      ;;Save working registers.
        PUSH    CX
        SUB     AX,AX                   ;;To start, result = 0.
        SUB     CH,CH                   ;;Read count into CX.
        MOV     CL,string_name
        LEA     BX,string_name+1        ;;Put first address in BX.
BLANKS:  CMP  BYTE PTR [BX],' '         ;;Scan past leading blanks.
        JNE     CHK_NEG
        INC     BX
        LOOP    BLANKS
;;
;;  Execute these instructions if the string starts with a
;;  minus sign.
;;
CHK_NEG:  CMP  BYTE PTR [BX],'-'
        JNE     CHK_POS
        INC     BX              ;;Increment the pointer.
        DEC     CX              ;;Decrement the count.
        CALL    CONV_AB         ;;Convert the string.
        JC      THRU
        CMP     AX,32768        ;;Is the number too small?
        JA      NO_GOOD
        NEG     AX              ;; No.  Complement the result.
        JS      GOOD
;;
;;  Execute these instructions if the first string character
;;  is not a minus sign.
;;
CHK_POS:  CMP  BYTE PTR [BX],'+'
        JNE     GO_CONV
        INC     BX                      ;;Increment the pointer.
        DEC     CX                      ;;Decrement the count.
GO_CONV:  CALL  CONV_AB                 ;;Convert the string.
        JC      THRU
        CMP     AX,32767                ;;Is the number too large?
```

```
      JA      NO_GOOD
GOOD:  CLC
      JNC     THRU
NO_GOOD:  STC                          ;; If so, set Carry.
THRU:  POP   CX                        ;;Restore registers.
      POP     BX
      JMP     SKIPIT
;;
;;  This subroutine performs the actual conversion.
;;
CONV_AB  PROC
      PUSH    BP                       ;;Save scratch registers.
      PUSH    BX
      PUSH    SI
      MOV     BP,BX                    ;;Put pointer in BP
      SUB     BX,BX                    ;; and clear BX.
RANGE:  CMP   BYTE PTR DS:[BP],'0'     ;;If character is not
      JB      NON_DIG                  ;; a digit,
      CMP     BYTE PTR DS:[BP],'9'
      JBE     DIGIT
NON_DIG:  STC                              ;; set Carry
      JC      END_CONV                     ;; and exit.
DIGIT:  MOV   SI,10                    ;;The character is a digit,
      PUSH    DX
      MUL     SI                       ;; so multiply AX by 10.
      POP     DX
      MOV     BL,DS:[BP]               ;;Fetch ASCII code,
      AND     BX,0FH                   ;; save only low bits,
      ADD     AX,BX                    ;; and update result.
      JC      END_CONV                 ;;Exit if result is too large.
      INC     BP                       ;;Otherwise increment BP
      LOOP    RANGE                    ;; and continue.
      CLC                              ;;When done, clear Carry.
END_CONV:  POP  SI                     ;;Restore registers.
      POP     BX
      POP     BP
      RET
CONV_AB  ENDP
SKIPIT:  NOP
      ENDM
```

BEEP MACRO

```
;;
;;  Beep the speaker for 1/2 second at a frequency of 1000 Hz.
```

```
;;
      SOUND   1000,50
       ENDM
```

BIN2$ MACRO string_name

```
      LOCAL   FILL_BUFF,CLR_DVD,NO_MORE
;;
;;  Convert a signed number in AX to a string in the data segment.
;;  String must be seven bytes long.  The first byte receives the
;;  character count.
;;
       PUSH   DX          ;;Save affected registers.
       PUSH   CX
       PUSH   BX
       PUSH   SI
       PUSH   AX
       LEA    BX,string_name+1  ;;BX points to first character.
       MOV    CX,6        ;;Fill buffer with spaces.
FILL_BUFF:  MOV  BYTE PTR [BX],' '
       INC    BX
       LOOP   FILL_BUFF
       MOV    SI,10       ;;Get ready to divide by 10.
       OR     AX,AX       ;;If value is negative,
       JNS    CLR_DIV
       NEG    AX          ;; make it positive.
CLR_DVD:  SUB  DX,DX      ;;Clear upper half of dividend.
       DIV    SI          ;;Divide AX by 10.
       ADD    DX,'0'      ;;Convert remainder to ASCII digit.
       DEC    BX          ;;Back up through buffer.
       MOV    [BX],DL     ;;Store character in string.
       OR     AX,AX       ;;Done?
       JNZ    CLR_DVD     ;; If not, get next digit.
       POP    AX          ;; Yes.  Get original value.
       OR     AX,AX       ;;Was it negative?
       JNS    NO_MORE
       DEC    BX          ;; Yes.  Store sign.
       MOV    BYTE PTR [BX],-'
NO_MORE:  MOV  string_name,6  ;;Record character count.
       POP    SI          ;;Restore registers.
       POP    BX
       POP    CX
       POP    DX
          ENDM
```

CLS MACRO

```
;;
;;  Clear the screen.
;;
     PUSH   AX           ;;Save affected registers.
     PUSH   BX
     PUSH   CX
     PUSH   DX
     MOV    CX,0         ;;Start at row 0, column 0.
     MOV    DH,24        ;;End at row 24
     MOV    DL,79        ;; and column 79.
     MOV    AH,6         ;;Select scroll-up option.
     MOV    AL,0         ;;Clear entire screen.
     MOV    BH,7
     INT    10H          ;;Issue Video I/O interrupt.
     POP    DX           ;;Restore flags.
     POP    CX
     POP    BX
     POP    AX
     ENDM
```

CR MACRO

```
;;
;;  Move cursor to beginning of current line.
;;
     PUSH   CX
     PUSH   DX
     POS                 ;;Read row number into DH.
     SUB    DL,DL        ;;Make column number 0.
     MOVE_CURSOR         ;;Move the cursor.
     POP    DX
     POP    CX
     ENDM
```

CRLF MACRO

```
;;
;;  Move cursor to beginning of next line.
;;
     CR                  ;;Carriage return.
     LF                  ;;Line feed.
      ENDM
```

DELAY MACRO minutes,seconds,hundredths

```
        LOCAL   SECS,MINS,HRS,CHECK,QUIT
;;
;;  Wait for a specified interval.
;;
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        READ_TIME           ;;Read current time.
        MOV     AH,CH       ;;Copy hours into AH.
        MOV     AL,CL       ;;      minutes into AL.
        MOV     BH,DH       ;;      seconds into BH.
        MOV     BL,DL       ;;      hundredths into BL.
;;
;;  Add the input values to the current time to get the target
;;  time.
;;
        ADD     AL,minutes
        ADD     BH,seconds
        ADD     BL,hundredths
;;
;;  Propagate any carryover.
;;
        CMP     BL,100
        JB      SECS
        SUB     BL,100
        INC     BH
SECS:   CMP     BH,60
        JB      MINS
        SUB     BH,60
        INC     AL
MINS:   CMP     AL,60
        JB      HRS
        SUB     AL,60
        INC     AH
HRS:    CMP     AH,24
        JNE     CHECK
        SUB     AH,AH
;;
;;  Wait for interval to elapse.
;;
CHECK:  READ_TIME           ;;Read the time again.
        CMP     CX,AX
```

```
        JA      QUIT
        JB      CHECK
        CMP     DX,BX
        JB      CHECK
QUIT:   POP     DX
        POP     CX
        POP     BX
        POP     AX
        ENDM
```

HOME MACRO

```
;;
;;  Move cursor to upper left-hand corner.
;;
        LOCATE  0,0
        ENDM
```

IN$ MACRO string_name

```
;;
;;  Read keyboard characters into the specified buffer in the
;;  data segment.
;;
        PUSH    DX
        LEA     DX,string_name
        IN$_DX
        POP     DX
        ENDM
```

IN$_DX MACRO

```
;;
;;  Read keyboard characters into a buffer in the data segment.
;;  (DS:DX) = Buffer address.
;;
        PUSH    AX
        MOV     AH,0AH      ;;Select keyboard read option.
        INT     21H         ;;Read keyboard.
        POP     AX
        ENDM
```

```
INKEY   MACRO

;;
;;  Read the next key into AL and display it.
;;
        PUSH    CX              ;;Save registers.
        PUSH    AX
        MOV     AH,1            ;;Select key option.
        INT     21H             ;;Read the key.
        POP     CX              ;;Restore AH.
        POP     AH,CH
        POP     CX              ;;Restore CX.
          ENDM

INKEY_I  MACRO

;;
;;  Read the next key into AL, but don't display it.
;;
        PUSH    CX              ;;Save registers.
        PUSH    AX
        MOV     AH,8            ;;Select key option.
        INT     21H             ;;Read the key.
        POP     CX              ;;Restore AH.
        POP     AH,CH
        POP     CX              ;;Restore CX.
            ENDM

KEY_$2BIN  MACRO

        LOCAL   TEMP,SAVE
;;
;;  Read a sequence of keyboard characters and convert them to
;;  a signed number in AX.
;;
        JMP     SAVE            ;;Skip temporary buffer.
TEMP    DB   7,8 DUP(?)
SAVE:   PUSH    DS              ;;Save affected registers.
        PUSH    CX
        MOV     CX,CS           ;;Make DS point to code segment.
        MOV     DS,CX
        IN$     TEMP            ;;Read string into TEMP.
        $2BIN   TEMP+1          ;;Make the conversion.
        POP     CX              ;;Restore registers.
        POP     DS
              ENDM
```

LF MACRO

```
;;
;;  Move the cursor to next line (same column position).
;;  If it is on the bottom line, move it to the top.
;;
     PUSH    CX
     PUSH    DX
     POS                 ;;Read row number into DH
     INC     DH          ;; and increment it.
     MOVE_CURSOR         ;;Move the cursor.
     POP     DX
     POP     CX
    ENDM
```

LOCATE MACRO row,col

```
;;
;;  Move the cursor to the specified row and column.
;;
     PUSH    DX
     MOV     DH,row
     MOV     DL,col
     MOVE_CURSOR
     POP     DX
        ENDM
```

MAKE_STACK MACRO stack_name

```
;;
;;  Set up a "standard" stack segment.
;;
stack_name  SEGMENT  PARA STACK 'STACK'
            DB   64 DUP('STACK   ')
stack_name  ENDS
            ENDM
```

MESSAGE MACRO string_name

```
;;
;;  Display the specified data string segment.  String must
;;  end with a $ character.
;;
     PUSH    DX
     LEA     DX,DS:string_name
```

```
     MESSAGE_DX
          ENDM
```

MESSAGE_DX MACRO

```
;;
;;  Display a message string in the data segment whose offset
;;  is in DX.  String must end with a $ character.
;;
     PUSH    AX
     MOV     AH,9          ;;Select string display option.
     INT     21H           ;;Display the string.
     POP     AX
             ENDM
```

MOVE_CURSOR MACRO

```
     LOCAL   OK,VS
;;
;;  Move the cursor to the row and column positions in DH and DL,
;;  respectively.  If the row exceeds 24 or the column exceeds 79,
;;  make it 0.
;;
      PUSH    AX           ;;Save affected registers.
      PUSH    BX
      CMP     DH,24        ;;Wrap-around inputs if needed.
      JNA     OK
      SUB     DH,DH
OK:   CMP     DL,79
      JNA     VS
      SUB     DL,DL
VS:   VIDEO_STATE          ;;Read display page into BH.
      MOV     AH,2         ;;Select cursor-moving option.
      INT     10H          ;;Move the cursor.
      POP     BX           ;;Restore flags and registers.
      POP     AX
               ENDM
```

POP_REGS MACRO reg_list

```
;;
;;  Retrieve registers from the stack.
;;
```

```
        IRP   reg,<reg_list>
        POP     reg
        ENDM
              ENDM
```

POS MACRO

```
;;
;;  Read the current row and column numbers into DH and DL,
;;  and the cursor mode into CH and CL.
;;
        PUSH    AX              ;;Save affected registers.
        PUSH    BX
        VIDEO_STATE             ;;Read display page into BH.
        MOV     AH,3            ;;Select cursor-reading option.
        INT     10H             ;;Read the cursor.
        POP     BX              ;;Restore registers.
        POP     AX
        ENDM
```

PRINT_AL MACRO

```
;;
;;  Display the character in AL, then advance the cursor.
;;
        PUSH    AX              ;;Save affected registers.
        PUSH    BX
        PUSH    CX
        PUSH    AX              ;;Preserve input value.
        VIDEO_STATE             ;;Read display page into BH.
        POP     AX              ;;Restore input value.
        MOV     CX,1            ;;Display just one character.
        MOV     AH,14           ;;Select display option.
        INT     10H             ;;Display the character.
        POP     CX              ;;Restore registers.
        POP     BX
        POP     AX
            ENDM
```

PRINT_NUMBER MACRO

```
        LOCAL   TEMP,SAVE
;;
;;  Display the signed contents of AX.
;;
```

```
      JMP     SAVE          ;;Skip temporary buffer.
TEMP   DB   7 DUP(?)
SAVE:   PUSH   DS           ;;Save affected registers.
      PUSH    CX
      MOV     CX,CS         ;;Make DS point to code segment.
      MOV     DS,CX
      BIN2$   TEMP          ;;Make the conversion.
      PRINT$  TEMP          ;;Display it.
      POP     CX            ;;Restore registers.
      POP     DS
          ENDM
```

PRINT$ MACRO string_name

```
;;
;;  Display a specified data segment string.
;;  First byte in the string holds the character count.
;;
      PUSH    DX
      LEA     DX,string_name
      PRINT$_DX
      POP     DX
         ENDM
```

PRINT$_DX MACRO

```
      LOCAL   STRIP,NEXT_C,NO_BLNK
;;
;;  Display the data segment string whose offset is in DX.
;;  First byte in the string holds the character count.
;;
      PUSH    AX            ;;Save affected registers.
      PUSH    CX
      PUSH    SI
      SUB     CX,CX         ;;Clear count register.
      MOV     SI,DX         ;;Put string pointer in SI.
      MOV     CL,[SI]       ;;Read character count.
      INC     SI            ;;Point to first character.
STRIP:  LODSB               ;;Skip leading blanks.
      CMP     AL,' '
      JNE     NO_BLNK
      LOOP    STRIP
NEXT_C:  LODSB              ;;Read character
NO_BLNK:  PRINT_AL          ;; and display it.
      LOOP    NEXT_C
```

```
        POP     SI              ;;Restore registers.
        POP     CX
        POP     AX
              ENDM
```

PUSH_REGS MACRO reg_list

```
;;
;;  Save registers on the stack.
;;
        IRP   reg,<reg_list>
        PUSH    reg
        ENDM
              ENDM
```

RAND MACRO limit

```
        LOCAL   STRIP
;;
;;  Generate a random integer between 0 and "limit," inclusive,
;;  and return it in AL.
;;
        PUSH    CX              ;;Save affected registers.
        PUSH    DX
        PUSH    AX
        MOV     AH,0            ;;Read the timer.
        INT     1AH
        MOV     AX,DX           ;;Move low count into AX.
        MOV     CL,limit        ;;Move limit into CL.
;;
;;  Strip enough high bits off the dividend (AX) to ensure against
;;  divide overflow.
;;
        MOV     DH,3FH          ;;Set up AND mask in DH.
        CMP     CL,64
        JAE     STRIP
        SHR     DH,1            ;;If limit < 64, strip off 3 bits.
        CMP     CL,32
        JAE     STRIP
        SHR     DH,1            ;;If limit < 32, strip off 4 bits.
        CMP     CL,16
        JAE     STRIP
        SHR     DH,1            ;;If limit < 16, strip off 5 bits.
        CMP     CL,0
        JAE     STRIP
```

```
     SHR     DH,1          ;;If limit < 8, strip off 6 bits.
STRIP:  AND   AH,DH        ;;Strip off the bits.
     DIV     CL            ;;Divide result in AX by limit in CL.
     MOV     AL,AH         ;;Put remainder in AL.
     POP     CX            ;;Restore AH.
     MOV     AH,CH
     POP     DX            ;;Restore other registers.
     POP     CX.
      ENDM
```

READ_TIME MACRO

```
;;
;;  Read the time into CH (hours), CL (minutes), DH (seconds),
;;  and DL (1/100 seconds).
;;
      PUSH    AX
      MOV     AH,2CH       ;;Select time-reading option.
      INT     21H          ;;Read the time.
      POP     AX
            ENDM
```

SET_DS MACRO dseg_name

```
;;
;;  Load the address of the data segment into DS.
;;
      PUSH    AX
      MOV     AX,dseg_name
      MOV     DS,AX
      POP     AX
         ENDM
```

SET_ES MACRO eseg_name

```
;;
;;  Load the address of the extra segment into ES.
;;
      PUSH    AX
      MOV     AX,eseg_name
      MOV     ES,AX
      POP     AX
         ENDM
```

```
SET_TIME  MACRO  hours,minutes,seconds,hsecs

;;
;;  Set the time.  Blank operands become zero.
;;
        PUSH    AX              ;;Save affected registers.
        PUSH    CX
        PUSH    DX
        MOV     CH,hours        ;;Read user values.
        MOV     CL,minutes
        MOV     DH,seconds
        MOV     DL,hsecs
        MOV     AH,2DH          ;;Select time set option.
        INT     21H             ;;Set the time.
        POP     DX              ;;Restore registers.
        POP     CX
        POP     AX
            ENDM

SHOW_AL  MACRO

;;
;;  Display the character in AL.  Keep the cursor where it is.
;;
        PUSH    AX              ;;Save affected registers.
        PUSH    BX
        PUSH    CX
        PUSH    AX              ;;Preserve input value.
        VIDEO_STATE             ;;Read display page into BH.
        POP     AX              ;;Restore input value.
        MOV     CX,1            ;;Display just one character.
        MOV     AH,10           ;;Select display option.
        INT     10H             ;;Display the character.
        POP     CX              ;;Restore registers.
        POP     BX
        POP     AX
            ENDM

SOUND  MACRO  freq,duration

;;
;;  Produce a tone of a specified frequency and duration.
;;  Frequency is in Hertz;; duration is in hundredths of a second.
;;
        MOV     DI,freq
```

```
      MOV     BX,duration
      SOUND_DI_BX
        ENDM

SOUND_DI_BX  MACRO

      LOCAL   WAIT,SPKR_ON
;;
;;  Produce a tone of a specified frequency and duration.
;;  Frequency in Hertz is in DI; duration in hundredths of a
;;  second is in BX.
;;
       PUSH    AX          ;;Save affected registers.
       PUSH    BX
       PUSH    CX
       PUSH    DX
       PUSH    DI
       MOV     AL,0B6H     ;;Write timer mode register.
       OUT     43H,AL
       MOV     DX,14H      ;;Timer divisor =
       MOV     AX,4F38H    ;; 1331000/Frequency.
       DIV     DI
       OUT     42H,AL      ;;Write Timer 2 count low byte.
       MOV     AL,AH
       OUT     42H,AL      ;;Write Timer 2 count high byte.
       IN      AL,61H      ;;Get current Port B setting
       MOV     AH,AL       ;; and save it in AH.
       OR      AL,3        ;;Turn speaker on.
       OUT     61H,AL
WAIT:  MOV   CX,2801       ;;Wait 10 milliseconds.
SPKR_ON:  LOOP  SPKR_ON
       DEC     BX          ;;Speaker-on count expired?
       JNZ     WAIT        ;; If not, keep speaker on.
       MOV     AL,AH       ;;Recover value of port.
       OUT     61H,AL
       POP     DI          ;;Restore registers.
       POP     DX
       POP     CX
       POP     BX
       POP     AX
         ENDM
```

TAB MACRO column

```
      LOCAL  DO_TAB,GET_BH
;;
;;  Move cursor to specified column (0 to 79).
;;  If it is beyond this position, move to specified column
;;  on the next line.
;;
      PUSH   AX           ;;Save affected registers.
      PUSH   BX
      PUSH   CX
      PUSH   DX
      POS                 ;;Read column number into DL.
      CMP    DL,column    ;;Cursor beyond this column?
      JBE    DO_TAB
      LF                  ;; If so, advance to next line
      POS                 ;;  and read new cursor position.
DO_TAB:  MOV  DL,column   ;;Read user's column number.
      CMP    DL,79        ;;Is it too large?
      JNA    GET_BH
      MOV    DL,79        ;; If so, use column 79.
GET_BH:  VIDEO_STATE      ;;Read active page into BH.
      MOV    AH,2         ;;Select cursor-moving option.
      INT    10H          ;;Move the cursor.
      POP    DX           ;;Restore registers.
      POP    CX
      POP    BX
      POP    AX
      ENDM
```

VIDEO_STATE MACRO

```
;;
;;  Read current mode into AL, number of screen columns into AH,
;;  and active page into BH.
;;
      MOV    AH,15        ;;Select video state option.
      INT    10H          ;;Read video state.
             ENDM
```

10

Object Libraries

In Chapter 9, we described how to create *macro libraries,* disk files that contain macro definitions. To use any macro in a library, simply INCLUDE the entire library in your program, and mention the macro's name. Therefore, macro libraries save you the bother of typing a macro in every program where you need it.

Version 2 of the Macro Assembler provides a similar facility for program modules. Specifically, it lets you create *object libraries,* disk files that contain object modules. To use any procedure in an object library, you CALL the procedure in your program and declare it external (with an EXTRN statement), as usual. Then, when you run the Linker, you link the library to the calling program module. The Linker automatically extracts the procedure you CALLed, just as if it were in a module by itself. Therefore, an object library saves you from having to remember which disk a procedure is on, as well as the inevitable disk-shuffling job that ensues. Now you must keep track of only one disk, the one that contains the object library.

10.1 Building an Object Library

The program that handles object libraries is the Library Manager (LIB.EXE) on the assembler disk. To create an object library, you must tell the Library Manager its name and specify the first object module you want to put in the library. To do this, put your assembler disk in drive A and the disk that contains the object module in drive B, then start the computer as usual. Get the B> prompt on the screen, then enter a command of the form

a:lib *libname* + *objmod;*

The Manager automatically supplies the extensions LIB and OBJ to the library and object module names.

For example, to create a library named OBJECT.LIB with SORT.OBJ as its first entry, enter

```
a:lib object + sort;
```

10.2 Operating on Object Libraries

Once you have created a library, you can add new modules to it by repeating the preceding command. For example, to add MULU32.OBJ to the OBJECT library, enter

```
a:lib object + mulu32;
```

The + here means "add to library." To delete a module from the library, use - (minus) instead of +. For example, to remove MULU32.OBJ, enter

```
a:lib object - mulu32;
```

You may also want to copy a module from the library, perhaps to add it to a different library. This requires the * operator. For example, to make a copy of SORT.OBJ, enter

```
a:lib object * sort;
```

After that, SORT is still in the library, but the disk contains a copy of it named SORT.OBJ.

The Library Manager also lets you combine the basic operations. The operator -+ replaces a library module with the contents of an object file of the same name. In other words, -+ *updates* a library. For example,

```
a:lib object -+ sort;
```

removes SORT from the OBJECT library, then replaces it with the contents of SORT.OBJ from the disk.

Similarly, the operator -* removes an object module from a library, but creates a copy of it on the disk. For example,

```
a:lib object -* sort
```

moves SORT from the OBJECT library to a new file called SORT.OBJ.

Getting a Directory of a Library

If you forget which object modules are in a library, you can get the Library Manager to generate a directory file. To do this, enter a command in this form:

a: lib *libname,libname.dir;*

To display the directory, enter

type *libname.dir*

Besides the names of the object modules, the directory will list the symbols that are declared PUBLIC in each one. This tells you the names by which your calling program must refer to procedures, variables, and equates in the library.

10.3 Using Object Libraries

As we mentioned at the beginning of this chapter, to use an object library, you must *link* it to the object module (or modules) that contains your program. The Linker can display a special prompt for this purpose. To obtain it, link using the form

a:link module,,NUL

When the computer shows

Libraries [.LIB]:

enter the name of your library, then press Return. For example, to link the OBJECT library to an object module called MAIN_PROG, enter **a:link main_prog,,NUL,** then

Libraries [.LIB]: **object**

If your program needs object modules from several different libraries, list them in response to the *Libraries* prompt. For example,

Libraries [.LIB]: **lib1 + lib2 + lib3**

11

Structured Programming

One of the best features of Macro Assembler Version 2 is that it lets you write *structured* assembly language programs. By "structured," we mean programs that contain high-level statements that perform loops, conditional jumps, and other control tasks that would otherwise take several assembly language instructions. In effect, the structure statements provide the same kinds of complex operations as BASIC. For example, one family of structure statements or *logic structure* does the same thing as an IF-THEN-ELSE statement in BASIC, while another provides a function similar to FOR-NEXT.

But structured programming involves more than simply using structure statements instead of instructions. Rather, it is a *philosophy* of software design that involves

- Top-down design. (Remember, this is an approach in which you build programs by starting with comments and add more and more detail in gradual steps.)
- Label-free programs.
- "GOTO"-free programs (no conditional nor unconditional jump instructions).
- Indentation to show levels of nesting in logic structures.
- Listings that are more self-documenting.

The purpose for doing all this is to increase the *quality* of assembly language programs. Quality improvement should be apparent in several areas:

- The developer will understand the logic of the program better.
- The listing will be more readable to someone trying to make changes when he or she is not the author.
- Because the program is more readily understandable, there should be fewer "bugs" in it.

- Reliability and serviceability are improved. This means the program is more apt to work correctly as designed, and future modifications can be made with more assurances that the changes will fit within the present design without introducing new bugs.
- Improved readability will help make the listing its own best documentation.

The feature that allows you to use structure statements is a program called *SALUT* (for Structured Assembly Language Utility) on the Macro Assembler Version 2.00 disk. What you do is enter your program, then run SALUT to convert the structure statements into standard assembly language instructions. From there, you assemble and link the program just as you always have.

This chapter describes SALUT's structure statements and gives examples and guidelines for using them. It also discusses the steps you must use to assemble and format structured programs.

11.1 Structure Statements and Logic Flow Structures

There are more than a dozen different structure statements, but you must use them in prescribed combinations, in what the Macro Assembler manual calls *logic flow structures.* The manual defines eight different logic flow structures, but we can treat them as three groups:

- *IF* is a decision-making structure. It makes the 8088 execute or skip a set of instructions, depending on whether a specified condition is satisfied. The IF structure is similar to IF-THEN in BASIC.
- *DO* is a repetition structure. It makes the 8088 loop through one or more sets of instructions until a specified condition is satisfied. The DO structure is similar to FOR-NEXT in BASIC.
- *SEARCH* is also a repetition structure, but it is used when you care *which* of several conditions caused the looping to stop. SEARCH executes a unique block of instructions for each exit condition.

Conditions for Logic Flow Structures

Table 11-1 lists the conditions you can use to control the IF, DO, and SEARCH structures. Note that except for NCXZ, they are the same conditions as those for the conditional transfer (Jx) instructions described in Chapter 3.

Table 11-1. Conditions for logic flow structures.

Condition	*Description*	*True if...*
A	Above	CF = 0 and ZF = 0
AE	Above or Equal	CF = 0
B	Below	CF = 1
BE	Below or Equal	CF = 1 or ZF = 1
C	Carry	CF = 1
CXZ	CX Register is Zero	(CX) = 0
E	Equal	ZF = 1
*G	Greater	ZF = 0 and SF = OF
*GE	Greater or Equal	SF = OF
*L	Less	SF ≠ OF
*LE	Less or Equal	ZF = 1 or SF ≠ OF
NA	Not Above	CF = 1 or ZF = 1
NAE	Not Above nor Equal	CF = 1
NB	Not Below	CF = 0
NBE	Not Below nor Equal	CF = 0 and ZF = 0
NC	No Carry	CF = 0
NCXZ	CX Register Not Zero	(CX) ≠ 0
NE	Not Equal	ZF = 0
*NG	Not Greater	ZF =1 or SF ≠ OF
*NGE	Not Greater nor Equal	SF ≠ OF
*NL	Not Less	SF = OF
*NLE	Not Less nor Equal	ZF = 0 and SF = OF
*NO	No Overflow	OF =0
NP	No Parity (Odd)	PF = 0
*NS	No Sign	SF = 0
NZ	Not Zero	ZF = 0
*O	Overflow	OF = 1
P	Parity (Even)	PF = 1
PE	Parity Even	PF = 1
PO	Parity Odd	PF = 0
*S	Sign	SF = 1
Z	Zero	ZF = 1

*Pertinent for signed (two's-complement) arithmetic.

A word of caution: CXZ and NCXZ are not always legal in structures. We discuss this more in Section 11.5.

11.2 The IF Structure

IF makes the 8088 either execute or skip a set of instructions, depending on whether a specified condition is true or false. Its basic form is

```
$IF cond
    A
$ENDIF
```

where *cond* is the test condition (one of the conditions in Table 11-1) and *A* is a block of instructions. If *cond* is true, the 8088 executes the instructions; otherwise, it skips to the statement that follows $ENDIF. (Note that this is the opposite of a conditional jump instruction. A conditional jump *skips* the instructions if the condition is true.) Figure 11-1(A) is a flowchart of the IF structure.

For example, to convert the number in BX to an absolute value, you could use:

```
CMP  BX,0          ;Value negative?
$IF  L
     NEG  BX       ;If so, make it positive
$ENDIF
```

The ELSE Option

As in BASIC, you can also insert an ELSE term to make the 8088 execute an alternate set of instructions if the condition is false. Then IF has the following form:

```
$IF   cond
      A   (Executed if cond is true)
$ELSE
      B   (Executed if cond is false)
$ENDIF
```

Figure 11-1(B) shows a flowchart of how the $IF-$ELSE-$ENDIF structure operates.

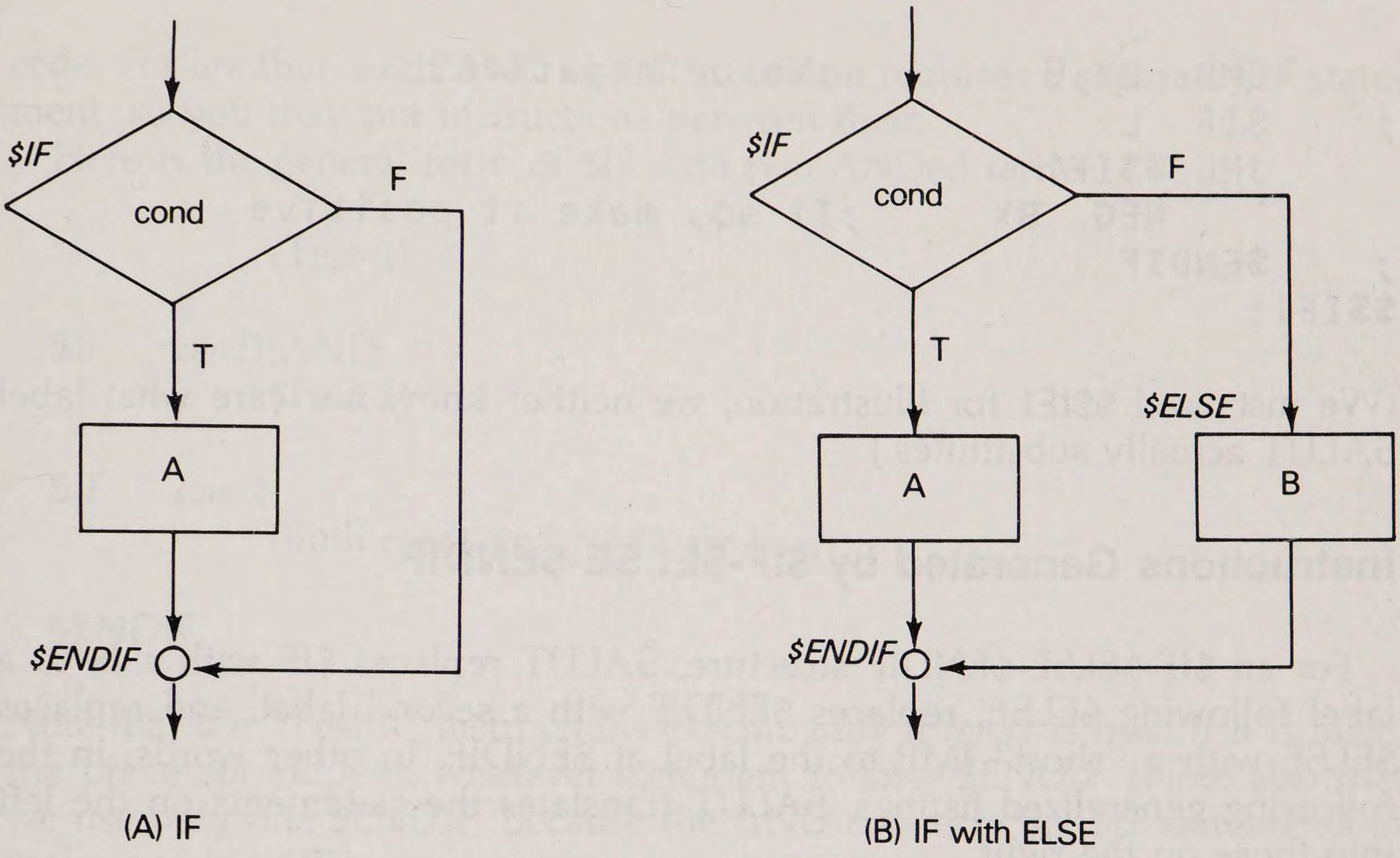


Figure 11-1. IF structures.

How the IF Structure Operates

The statements $IF, $ELSE, and $ENDIF are actually commands for SALUT. When you run SALUT, it converts these statements to comment lines (so the assembler will ignore them), then generates their assembly language definitions. After that, the assembler (even the *Small* version) can assemble the program as if $IF, $ELSE, and $ENDIF were never there. In the following sections we will describe how SALUT converts an $IF-$ENDIF structure, then how it converts an $IF-$ELSE-$ENDIF structure.

Instructions Generated by $IF-$ENDIF

When SALUT encounters an $IF, it generates a conditional transfer (Jx) instruction that makes the 8088 skip the statements between $IF and the next $ENDIF. Note the word "skip" here. This tells you that the assembler will substitute a Jx instruction that is the *reverse* of what the $IF operand calls for! Hence, it will replace *$IF Z* with *JNZ* and *$IF A* with *JNA*. Of course, since each conditional transfer instruction requires a target, SALUT inserts a label after $ENDIF. For instance, after SALUT has processed $IF and $ENDIF, our preceding example would look like this:

```
        CMP   BX,0              ;Value negative?
;       $IF   L
        JNL $$IF1
            NEG   BX            ;If so, make it positive
;       $ENDIF
$$IF1:
```

(We just used $$IF1 for illustration; we neither know nor care what label SALUT actually substitutes.)

Instructions Generated by $IF-$ELSE-$ENDIF

For an $IF-$ELSE-$ENDIF structure, SALUT replaces $IF with a Jx to a label following $ELSE, replaces $ENDIF with a second label, and replaces $ELSE with a "short" JMP to the label at $ENDIF. In other words, in the following generalized listings, SALUT translates the statements on the left into those on the right.

```
$IF   cond                      JNcond $$IF1
  A                               A
$ELSE                           JMP SHORT $$EN1
                              $$IF1:
    B                             B
$ENDIF                        $$EN1:
```

Operand Options

The general forms for $IF and $ELSE are:

```
$IF     cond[,AND/OR][,LONG]
$ELSE   [LONG]
```

where *AND* and *OR* let you specify multiple conditions and *LONG* lets you use long blocks of instructions within the structure.

AND and OR Options

So far, we have discussed IF structures that depend only on a single condition. However, as in BASIC, you can make IF consider a combination of conditions with either an AND or an OR.

There are some differences between these assembly language conditionals and those in BASIC, however. For one thing, the processor considers each term in the order it appears; there are no parentheses to tell it how to pre-

cede. For another, each AND or OR condition requires a separate $IF statement, so you may put instructions between them.

Here is the general form of $IF with two ANDed terms:

```
        ..      (Test 1)
        ..
$IF     cond1,AND
        ..      (Test 2)
        ..
$IF     cond2
        ..      (Both cond1 and cond2 are true)
        ..
$ENDIF
```

Note that the "Test 2" instructions execute only if *cond1* is true. If it is false, the processor skips to whatever statement follows $ENDIF. (Note also that we use only one $ENDIF, because the AND connects the $IF statements to make one big $IF.)

The instructions that follow the first $IF statement are used to set up the flag(s) for the second $IF. For example, to print "Weight okay." if WEIGHT is between 31 and 34, use:

```
        CMP   WEIGHT,31
$IF     AE,AND
        CMP   WEIGHT,34
$IF     BE
        (Print "Weight okay.")
$ENDIF
```

Of course, you could also include a $ELSE term to do something (say, print "Weight out of range.") if either condition is not satisfied.

Here is the general form of $IF with two ORed terms:

```
        ..      (Test 1)
        ..
$IF     cond1,OR
        ..      (Test 2)
        ..
$IF     cond2
        ..      (cond1 or cond2 is true)
        ..
$ENDIF
```

Here, the processor executes the "Test 2" instructions only if the *cond1* test fails.

LONG Jumps

Because SALUT replaces $IF and $ELSE statements with Jx and JMP SHORT instructions, a block can be no longer than 127 bytes. In 127 bytes, you can put about 50 or 60 instructions, so this is usually no limitation. However, if your IF structure includes a lot of instructions, you must declare $IF or $ELSE *LONG*.

For example, you could use

```
$IF   A,LONG
```

LONG makes SALUT replace *$IF A* with

```
              JA $$XL1
              JMP $$IF1
      $$XL1:
```

rather than with *JNA $$IF1*.

11.3 The DO Structure

DO makes the 8088 repeat blocks of instructions until a specified condition is satisfied. The DO structure is available in three forms:

- *DO UNTIL* repeats a block of instructions until a condition at the end becomes true.
- *DO WHILE* executes a block of instructions until a condition at the start becomes true. (In other words, it "does" the block while the condition is false.) If the condition is initially true, it skips the block altogether.
- *DO COMPLEX* is a combination of DO UNTIL and DO WHILE that involves two blocks of instructions. It executes the first block, then tests the condition. If the condition is true, it executes the second block and starts again; otherwise, it skips the second block and exits.

Note: The words UNTIL, WHILE, and COMPLEX simply help us distinguish between the three forms of the DO structure; *don't* enter these words into your programs. (However, IBM *has* provided an alternate form of DO COMPLEX in which you must type the word COMPLEX, but we won't discuss it here.)

DO UNTIL

DO UNTIL structures have the general form

```
$DO
        A
$ENDDO  cond
```

where *cond* is the condition that makes the looping stop. Because the processor tests *cond* at the end of the loop, the instructions always execute at least once. Figure 11-2(A) is a flowchart that shows how DO UNTIL operates.

The DO UNTIL structure is often used to repeat instructions based on a count value in CX. For example:

```
MOV     CX,10
$DO
        ..      (These instructions will be repeated 10 times.)
        ..
$ENDDO  LOOP
```

Here, LOOP decrements CX and loops to $D0 if CX is nonzero. The equivalent sequence in BASIC is:

```
FOR X=10 TO 1 STEP -1
..
..
NEXT X
```

DO WHILE

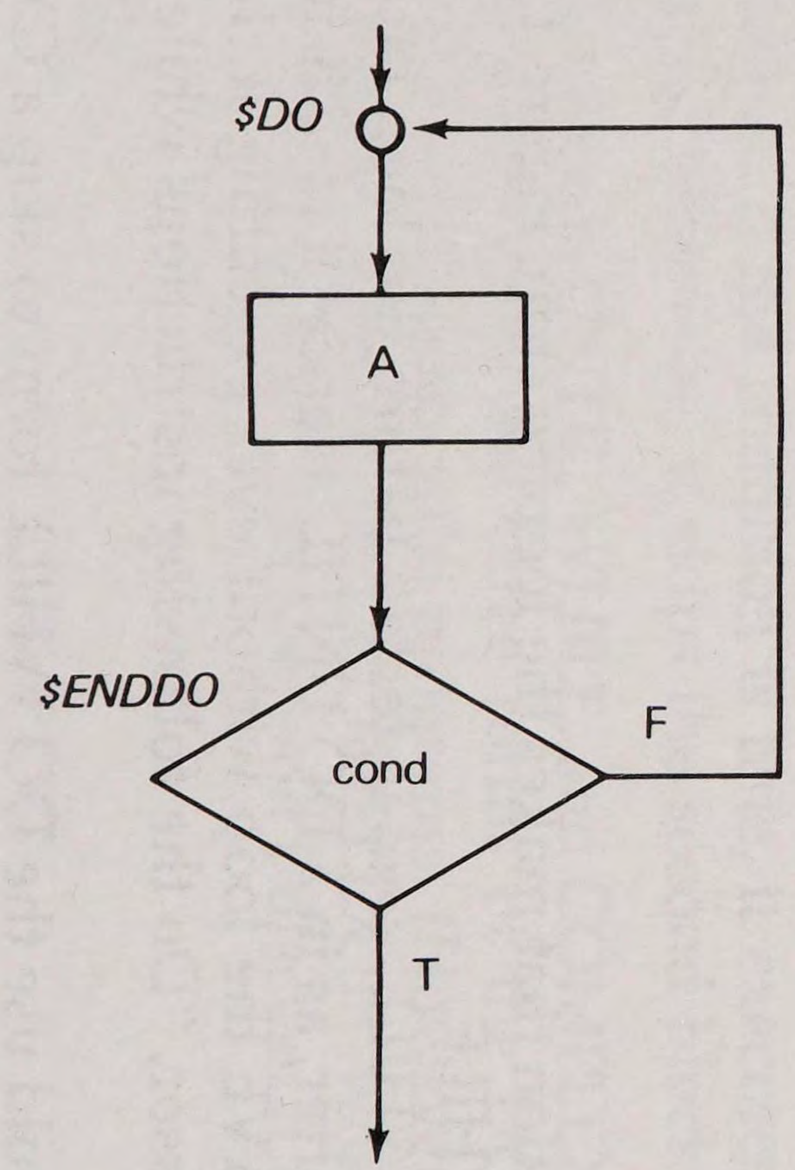
The DO WHILE structure has the general form

```
$DO
$LEAVE  cond
        A
$ENDDO
```

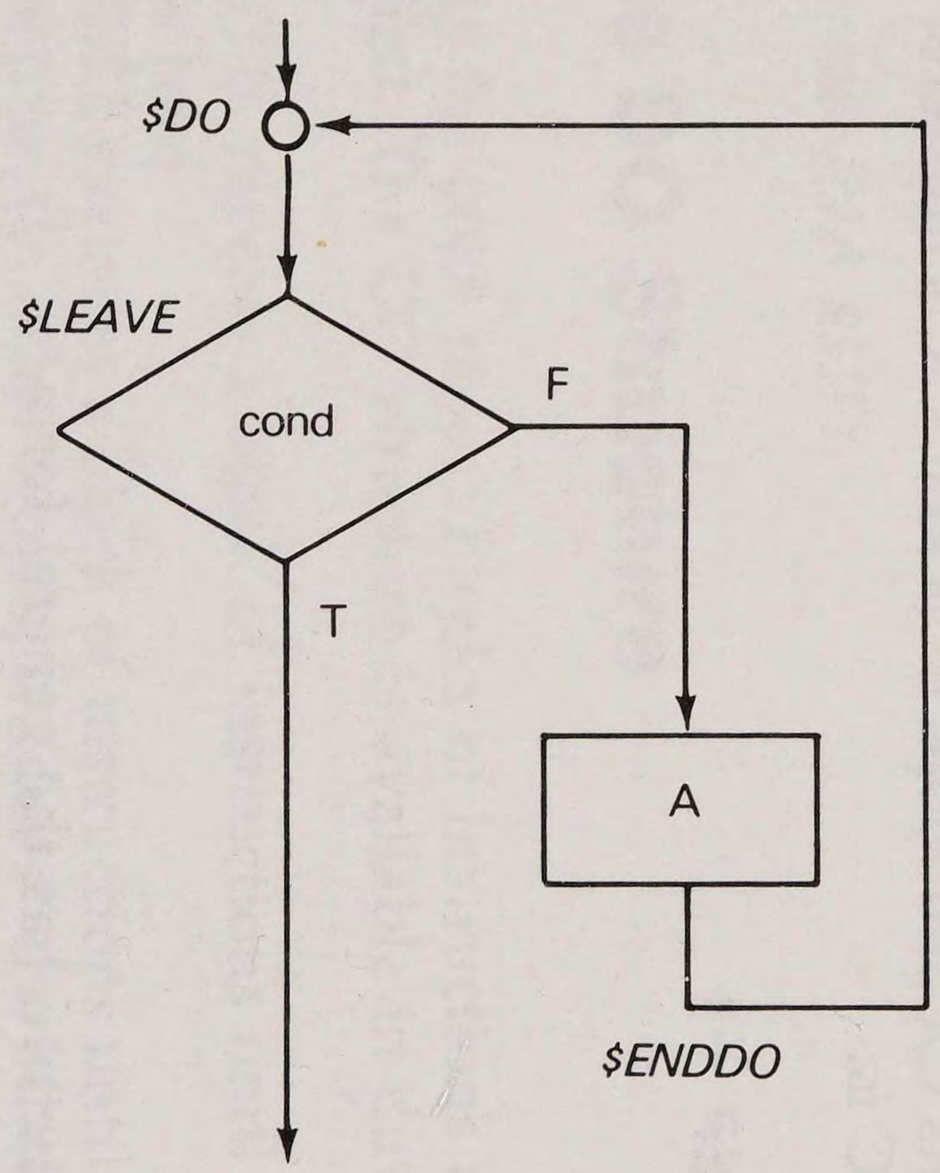
where *cond* is the condition that makes the looping stop. Figure 11-2(B) shows a flowchart for DO WHILE.

Note that in a DO WHILE, *cond* is tested before the loop instructions are executed, rather than after, as in a DO UNTIL. Hence, if *cond* is initially false, the processor will LEAVE the loop without ever executing it. In effect, DO WHILE tells the processor, "Do the following instructions while *cond* has not yet happened."

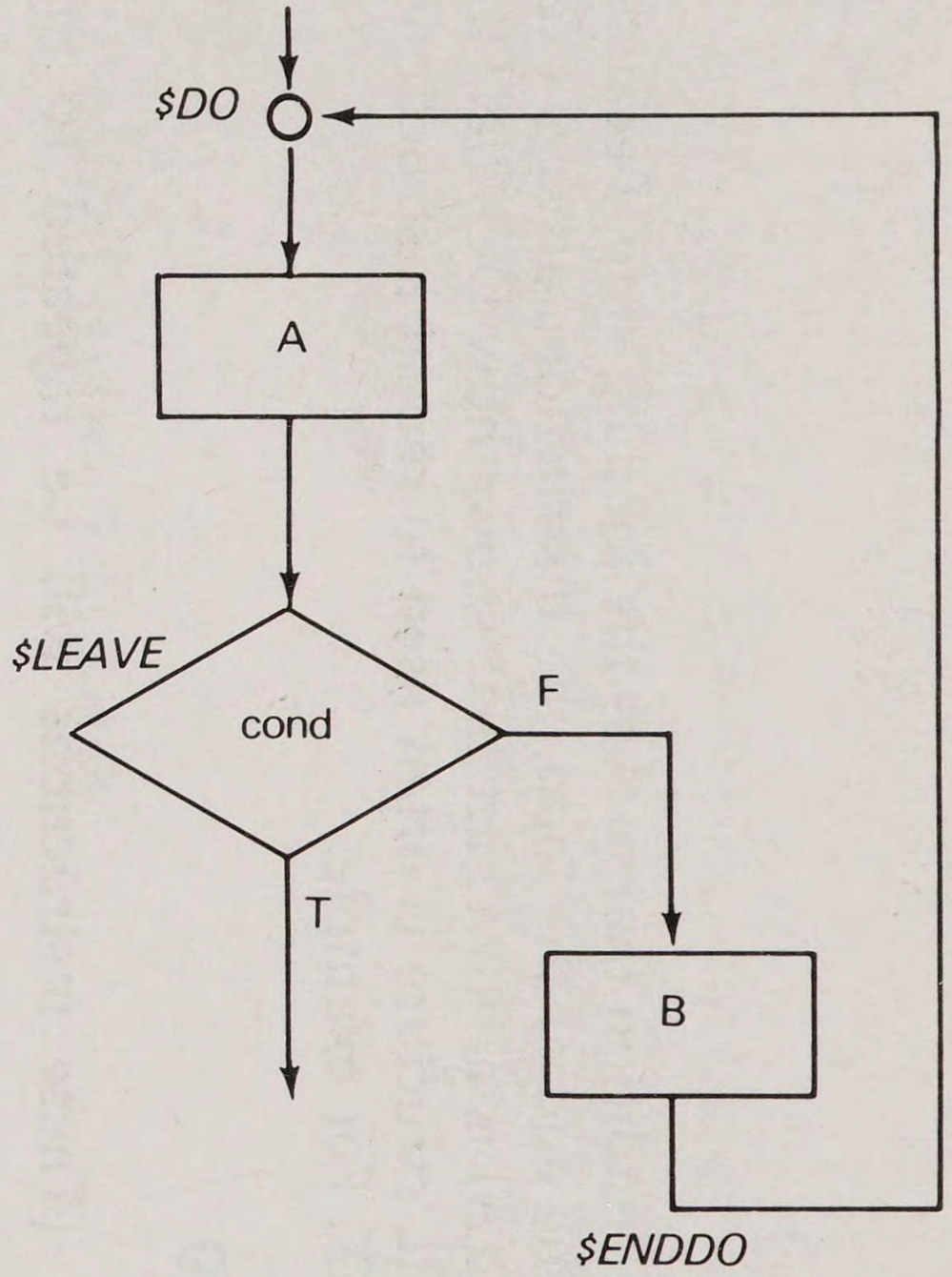
For example, you could use the DO WHILE form to skip a CX-based loop operation if CX is already zero, as in



(A) DO UNTIL



(B) DO WHILE



(C) DO COMPLEX

Figure 11-2. DO structures.

```
$DO
$LEAVE   CXZ
      ..
      ..
$ENDDO   LOOP
```

DO COMPLEX

The DO COMPLEX structure is similar to DO WHILE, but it lets you do something both before and after the LEAVE test—thus making the exit in the middle of the loop. The general form is

```
$DO
        A
$LEAVE  cond
        B
$ENDDO
```

As before, *cond* is the condition that makes the looping stop. Figure 11-2(C) shows a flowchart for DO COMPLEX.

Here, the A block instructions establish *cond*, while the B block instructions do whatever a false *cond* implies. For example, to delay a program until the user has entered a correct password, use the following approach:

```
$DO
      (Display "Please enter your password."
      (Check password. If valid, make CF = 1.)
$LEAVE  C
      (Display "Invalid password. Please try again.")
$ENDDO
```

Operand Options

Like $IF in Section 11.2, $ENDDO and $LEAVE have *AND* and *OR* options that let you combine conditions and a *LONG* option that lets you use long blocks of instructions. $ENNDO also has *LOOP* options that save you the trouble of decrementing CX. The general forms are:

```
$ENDDO   [cond[,AND/OR]][,LONG]
$ENDDO   [LOOPx][,LONG]
$LEAVE   cond[,AND/OR][,LONG]
```

LOOP Options

For counting loops based on CX, SALUT lets you omit the DEC CX instruction by using the form *$ENDDO LOOP*. That is, instead of

```
$DO
      ..
      ..
      DEC   CX
$ENDDO   CXZ
```

you can use the simpler form

```
$DO
      ..
      ..
$ENDDO   LOOP
```

This makes SALUT replace $DO with a label and $ENDDO with a *LOOP label* instruction. Recall from Section 3.7 that LOOP decrements CX and makes the 8088 transfer to the label if CX is not zero.

To include the state of the Zero Flag (ZF) in the loop decision, use one of the following forms:

- *$ENDDO LOOPE* decrements CX, then transfers if CX is not zero and ZF is 1. This locates the first non-zero result in a series of operations.
- *$ENDDO LOOPNE* decrements CX, then transfers if CX is not zero and ZF is zero. This locates the first zero result in a series of operations.
- *$ENDDO LOOPZ* is an alternate form of *$ENDDO LOOPE*.
- *$ENDDO LOOPNZ* is an alternate form of *$ENDDO LOOPNE*.

AND and OR Options

The AND option stops the looping when several conditions are satisfied, while OR stops it when *any* of several conditions is satisfied. For example, here is the general form of a DO structure that stops looping when the user answers "No" to a prompt by pressing either *N* or *n*:

```
$DO
      ..
      ..
      (Display "Continue? (Y/N)")
      (Read user's response into ANSWER)
      CMP   ANSWER,"N"
```

```
$ENDDO  E,OR
      CMP  ANSWER,"n"
$ENDDO  E
```

11.4 The SEARCH Structure

A SEARCH structure is similar to a DO structure, but SEARCH provides two separate ways to leave a loop: on "success" or "failure." This is handy for applications where you care about *which* condition made the looping stop. For example, you would want to know whether an I/O operation ended successfully or aborted due to an error.

Like DO, the SEARCH structure is available in three forms:

- *SEARCH UNTIL* repeats a block of instructions until either of two conditions at the end becomes true.
- *SEARCH WHILE* repeats a block of instructions while both conditions are false. If either condition is initially true, it does not execute the block at all.
- *SEARCH COMPLEX* is a combination of SEARCH UNTIL and SEARCH WHILE that involves a block of instructions preceding the condition tests and another following them. It executes the first block, then tests the conditions. If either condition is true, it exits; otherwise, it executes the second block and starts again.

Note: As with DO, the words UNTIL, WHILE, and COMPLEX are for description only; *don't* type them into your programs.

SEARCH UNTIL

The SEARCH UNTIL structure has the general form

```
$SEARCH
        A       (Main block)
$EXITIF     cond1
        B       (Executes, then exits, if cond1 is true)
$ORELSE
        C       (Executes if cond1 is false)
$ENDLOOP cond2
        D       (Executes, then exits, if cond2 is true)
$ENDSRCH
```

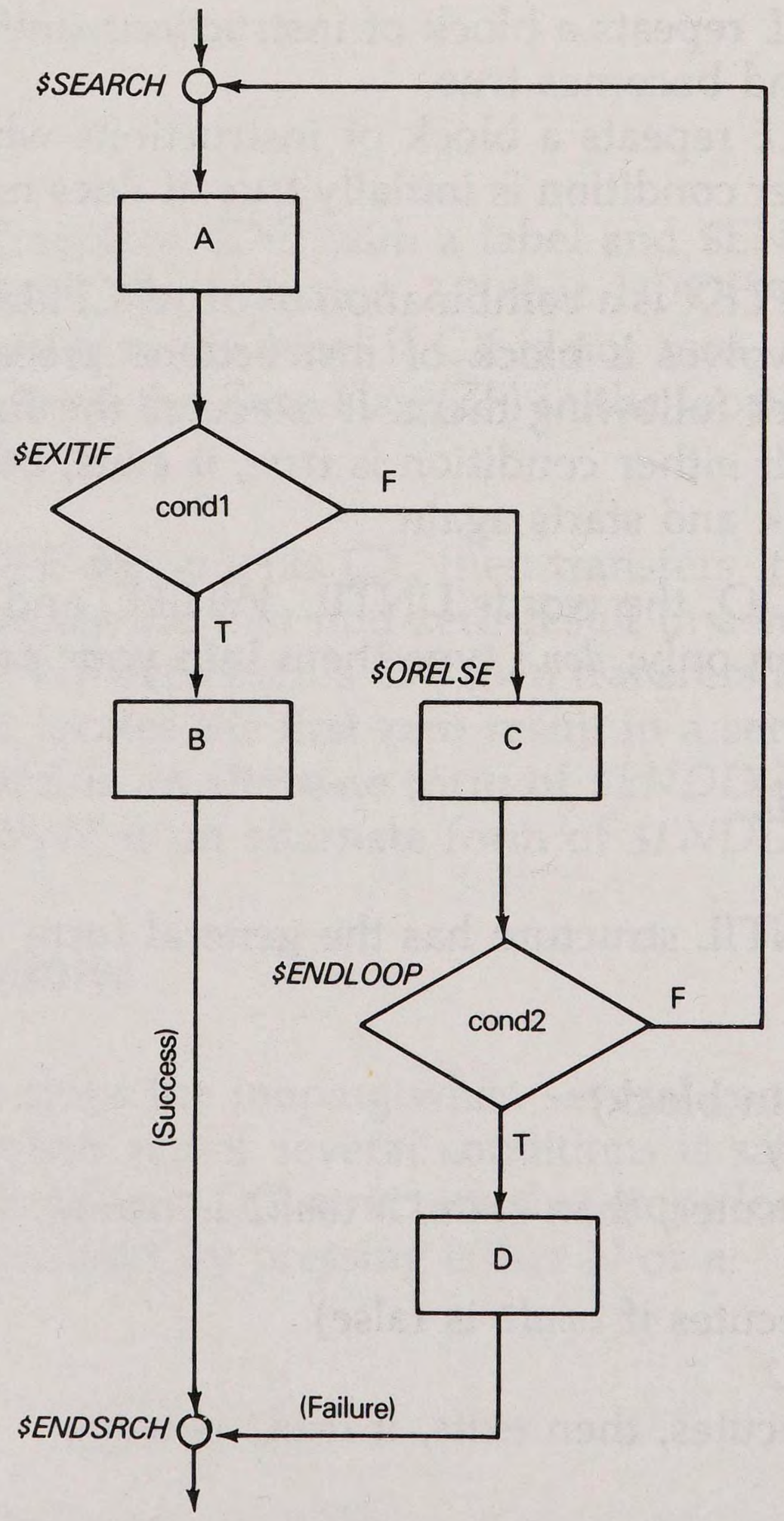
where *cond1* and *cond2* are the conditions that can make the looping stop. Here, *cond1* true indicates success (you found what you were searching for),

while *cond2* true indicates failure. Figure 11-3(A) shows the flowchart for SEARCH UNTIL.

Note that the main block, A, always executes at least once. Blocks B and D execute upon the success or failure of the operation, respectively (you might use them to print messages), while block C contains instructions that set up the flags for the $ENDLOOP test.

For example, in an educational program, use SEARCH UNTIL to give a student three chances to select the correct answer to a multiple-choice question. The general form might look like this:

```
        MOV   CX,3          ;Set up the count.
        (Display the question.)
$SEARCH
        (Read the student's answer.)
```



(A) SEARCH UNTIL

Figure 11-3(A). SEARCH UNTIL flowchart.

```
      (Make ZF = 1 if answer is correct.)
$EXITIF   Z
      (Display "Congratulations! You are right.")
$ORELSE
      (Display "Sorry, wrong answer. Try again.")
$ENDLOOP  LOOP
      (Display "No, all three answers were wrong.")
      (Display "The correct answer is " and give answer.)
$ENDSRCH
```

SEARCH WHILE

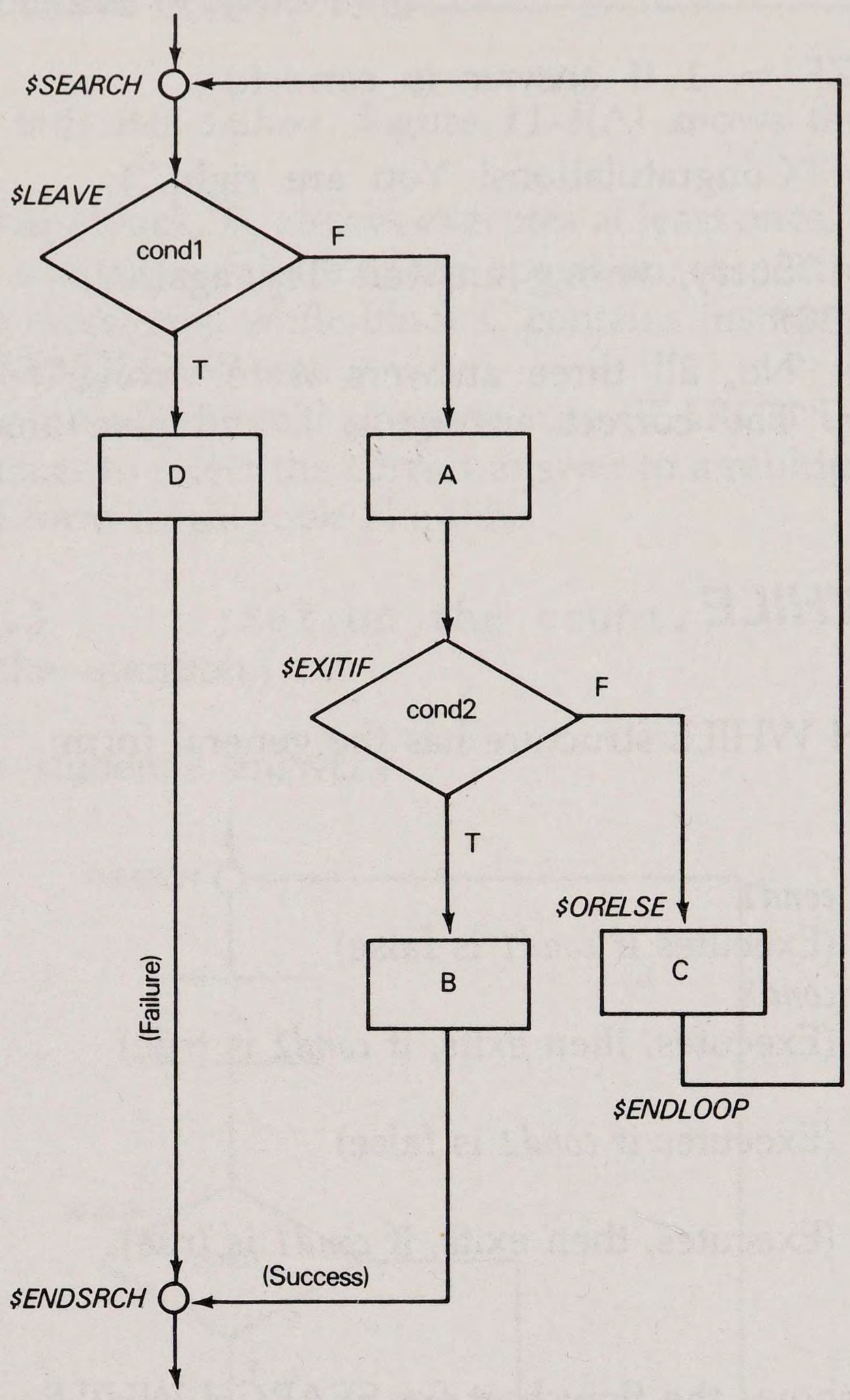
The SEARCH WHILE structure has the general form

```
$SEARCH
$LEAVE      cond1
      A     (Executes if cond1 is false)
$EXITIF     cond2
      B     (Executes, then exits, if cond2 is true)
$ORELSE
      C     (Executes if cond2 is false)
$ENDLOOP
      D     (Executes, then exits, if cond1 is true)
$ENDRCH
```

Figure 11-3(B) shows the flowchart for SEARCH WHILE.

Because SEARCH WHILE tests at the very beginning, you can use it to bypass the entire structure if the initial conditions are not satisfied. For example, the following searches for the first zero in a table, and "LEAVEs" immediately if the table is empty:

```
      MOV   CX,TABLE_LENGTH      ;Set up the count.
      MOV   BX,0                 ;Point to first element.
$SEARCH
$LEAVE   CXZ
      CMP   TABLE[BX],0          ;Check for zero.
$EXITIF   E
      (Display "Zero found.")
$ORELSE
      INC   BX                   ;Point to next element.
$ENDLOOP  LOOP
      (Display "Zero not found.")
$ENDSRCH
```



(B) SEARCH WHILE

Figure 11-3(B). SEARCH WHILE flowchart.

SEARCH COMPLEX

The SEARCH COMPLEX structure has the general form

```
$SEARCH
        A
$EXITIF     cond1
        B       (Executes, then exits, if cond1 is true)
$ORELSE
        C       (Executes if cond1 is false)
$LEAVE      cond2
        D       (Executes if cond2 is false)
$ENDLOOP
```

E (Executes, then exits, if *cond2* is true)
$ENDSRCH

Figure 11-3(C) shows the flowchart for SEARCH COMPLEX. Note that SEARCH COMPLEX does the same thing as SEARCH UNTIL, but executes an additional block (D) if *cond2* is false.

Operand Options

Some SEARCH structure statements have options like those we encountered for IF and DO. Their general forms are:

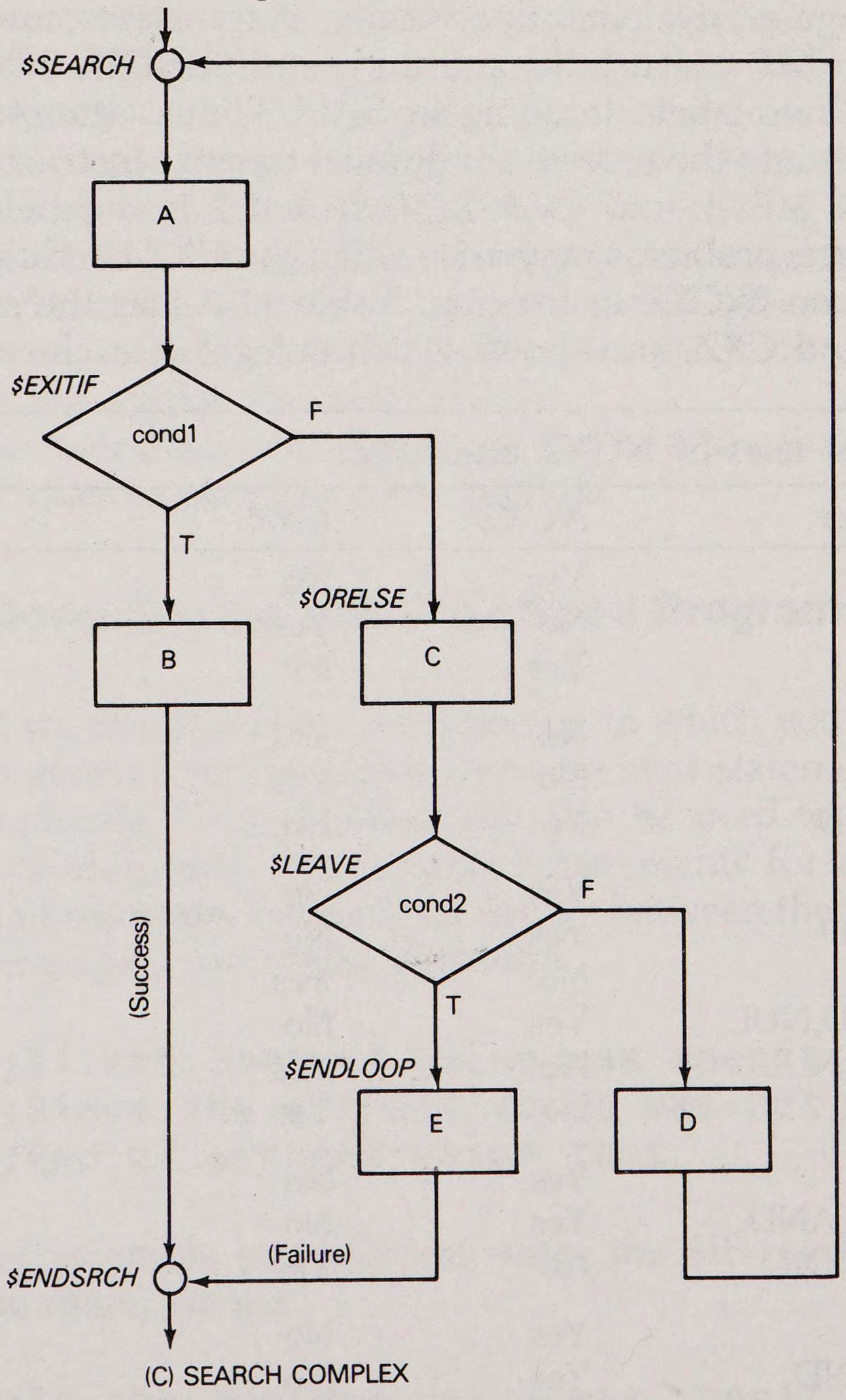


Figure 11-3(C). SEARCH COMPLEX flowchart.

```
$EXITIF     cond[,AND/OR][,LONG][,NUL]
$ORELSE     [LONG]
$ENDLOOP    [cond[,AND/OR]][,LONG]
$ENDLOOP    [LOOPx][,LONG]
$LEAVE      cond[,AND/OR][,LONG]
```

The only new option is NUL, which you can use with $EXITIF if you omit the B ("success") block from a SEARCH structure. Of course, with B gone, you can also omit $ORELSE. The structure will still work if you omit the B block, but NUL makes it execute faster.

11.5 Restrictions on Using NCXZ and CXZ

As we mentioned earlier, SALUT translates structure statements into their assembly language equivalents. Specifically, it translates conditional statements into Jx or JMP instructions and translates $ENDIF, $DO, $SEARCH, and $ENDSRCH into labels. In doing so, SALUT must often translate a conditional statement into the reverse conditional transfer instruction. For example, it translates *$IF A* into *JNA L1* (where L1 is a label that replaces $ENDIF). However, problems may arise when the CXZ condition is involved, because there is no JNCXZ instruction. Table 11-2 lists the statements that can use NCXZ and CXZ, and shows which is legal in each case.

Table 11-2. Legal uses of NCXZ and CXZ.

Structure Statement	*NCXZ*	*CXZ*
$IF cond	Yes	No
$IF cond,OR	No	Yes
$IF cond,AND	Yes	No
$LEAVE cond	No	Yes
$LEAVE cond,OR	No	Yes
$LEAVE cond,AND	Yes	No
$EXITIF cond	Yes	No
$EXITIF cond,AND	Yes	No
$EXITIF cond,OR	No	Yes
$EXITIF cond,AND,NUL	Yes	No
$EXITIF cond,OR,NUL	No	Yes
$EXITIF cond,NUL	No	Yes
$ENDLOOP cond	Yes	No
$ENDLOOP cond,AND	Yes	No
$ENDLOOP cond,OR	No	Yes
$ENDDO cond	Yes	No
$ENDDO cond,AND	Yes	No
$ENDDO cond,OR	No	Yes

11.6 Preparing Structured Programs

The Procedure

You can prepare structured programs in about the same way you prepare ordinary programs, except you must pass the source through SALUT, to translate the structures into standard assembly language instructions. SALUT will also format your source program into evenly-spaced columns. The steps are as follows:

1. Prepare the program using your word processor, editor, or EDLIN. Give it a name of the form *filename.SAL*, where SAL stands for Structured Assembly Language.

   Important: When entering the program, start all stand-alone comments and labels in column 1, but start structure statements, unlabeled instructions, and unlabeled pseudo-ops in some column other than column 1. For example, you might start them in column 2 by pressing the space bar or (using EDLIN) in column 9 by pressing Tab.
2. Run the SAL file through SALUT to generate a formatted source file (*filename.SAL*) and a temporary intermediate file (*filename.ASM*). SALUT renames your original, unformatted source file *filename.BAK*.
3. Assemble the source file using MASM.
4. Delete the temporary .ASM file and, if you wish, the .BAK file.
5. Use the Linker to generate a run module.

Using Top-Down Design with Structured Programs

In Chapter 2 we discussed top-down design, in which you build a program by entering its general outline (often just comment statements), then insert more detail gradually. This approach can also be used to build structured programs. To do this, enter the command statements for a logic structure, then go back and insert the instructions that go between them. To build an IF structure, for example, you might start with

```
$IF         ;If the correct value was specified
$ELSE       ;Since the correct value was not specified
$ENDIF      ;End of correct value test
```

This will not assemble properly yet, since the $IF requires a condition. Adding the condition, we get

```
$IF E       ;If the correct value was specified
```

```
$ELSE       ;Since the correct value was not specified
$ENDIF      ;End of correct value test
```

We now have a condition "equal," which must follow a comparison test. Hence, insert the test and get

```
CMP AX,100 ;Check for desired value
$IF E      ;If the correct value was specified
$ELSE      ;Since the correct value was not specified
$ENDIF     ;End of correct value test
```

Note that what we have here will assemble correctly and is actually executable. Therefore, we could leave this part of the program for awhile and come back later to fill in the "true" and "false" legs of the test. After all, we have enough here to document what must be done; we can add the details later.

The important point in this example is that I have added *comments* along with the instructions. There is no better time to document a program than when writing it, because you will never understand it better than you do then. Don't ever say to yourself "I will comment it later," because you probably won't. And even if you try, when "later" comes, chances are you won't remember what you were trying to accomplish.

Using SALUT

To use SALUT, get the B> prompt on the screen, then enter a command of the form

B>**a:salut** *filename*

(SALUT assumes the file has the extension SAL unless you specify otherwise.)

When SALUT finishes, the disk in drive B will contain three files: the original source (*filename.BAK*), a formatted version of the source (*filename.SAL*), and a temporary intermediate file (*filename.ASM*). You can now assemble *filename.ASM* as usual, then link *filename.OBJ* to produce the run module, *filename.EXE*.

Batch File for SALUT

Generally, once you have finished with SALUT, you won't want your original source file (now named BAK). And once you have assembled, you won't want the ASM file (because if the program contains errors, you will correct the SAL version rather than the ASM version). Therefore, you will want to

erase these two files. To save the bother of erasing them manually, create a *batch file* that runs SALUT and the Macro Assembler, then erases the BAK and ASM files. To do this (assuming you want to name the batch file ASMSAL.BAT), enter the following, and press Return after each line:

```
B>copy con: a:asmsal.bat
    a:salut %1
    erase %1.bak
    a:masm %1,,;
    erase %1.asm
    (Press F6)
```

To run this batch file, enter a command of the form

B>**a:asmsal** *filename*

Incidentally, note that the fourth command automatically produces a source listing (LST) file. If you don't want one, create another batch file that has *a:masm %1;* instead.

How SALUT Reformats Programs

Unless you tell it otherwise, SALUT reformats programs as follows:

1. *Labels* and *stand-alone comments* start at column 1.
2. *Structure statements* and *instruction mnemonics* start at column 9.
3. *Operands* start at column 17.
4. *Comments* start at column 41.
5. *Statements inside structures* are indented four spaces.

You can specify other settings if you like. Refer to the *Macro Assembler* manual for details.

To see how this formatting works, enter the following sequence in a program:

```
MOV CX,4 ;Perform four operations.
MOV BX,0 ;Point to first element.
$DO
MOV DEST[BX],0 ;Store zero in next byte.
INC BX ;Point to next element.
$ENDDO LOOP
```

After SALUT has formatted it, the sequence is arranged as follows:

```
        MOV     CX,4                    ;Perform four operations.
        MOV     BX,0                    ;Point to first element.
        $DO
            MOV     DEST[BX],0          ;Store zero in next byte.
            INC     BX                  ;Point to next element.
        $ENDDO  LOOP
```

Indenting Programs Using SALUT

Note that SALUT's reformatting capability is separate from its structure-processing capability. Hence, even if you never use structures (shame on you!), you can still run SALUT to indent your assembly language programs. To do this, enter your program any old way at all (say, put a space or tab between fields), then save the program and give a command of the form

B>**a:salut** *filename*.**asm,***filename*.**asm,nul**

SALUT indents *filename*.ASM and renames your unformatted version *filename*.BAK.

12

8087 Math Coprocessor

The Intel 8087 Math Coprocessor is a chip that performs complex mathematical calculations. It is designed to extend the arithmetic capability of an 8088 or 8086 microprocessor. In fact, the IBM Personal Computer (and its "work-alikes") provides an extra socket for an 8087.

As you know, the 8088 can already do some arithmetic—but not much. It can operate on only five-digit (two-byte) integers, and provides only the four basic functions: add, subtract, multiply, and divide. On the other hand, the 8087 can perform a wide variety of arithmetic, logarithmic, and trigonometric functions on integer and real numbers up to 18 digits long. And because the 8087's instructions are built into hardware, it can produce a dramatic speed improvement over doing these operations with the 8088. For example, it might take an 8088 program 2100 microseconds to do a 64-bit multiplication, but the 8087 can do the same thing in only 30 microseconds!

The 8087 is called a *coprocessor* because it acts in combination with the main processor (the 8088) to execute programs. When you run a program that is designed to use the 8087, the 8088 executes the instructions that it recognizes and the 8087 executes those that *it* recognizes. Each intercepts its own instructions. (Think of a construction foreman who has two helpers, an Italian and a Spaniard. The Italian worker will perform only orders the foreman gives in Italian, while the Spanish worker will perform only those given in Spanish.)

Programs written in a high-level language such as BASIC or Pascal use the 8087 automatically. Assembly language, however, requires you to either insert 8087 instructions with the 8088's ESC (escape) instruction or buy an assembler that recognizes the 8087 instruction set, such as the MACRO Assembler program in Version 2 of IBM's Macro Assembler package.

This chapter gives you a brief overview of the 8087 NDP. For full details, refer to the Macro Assembler manual or order a copy of Richard Startz' excellent book *8087 Applications and Programming* (Brady Communications Company, Inc., 1983). As a further aid, the Version 2 Macro Assembler disk

contains a file called *FLINS.ASM* that contains examples of most 8087 instructions.

12.1 Internal Registers

The 8087 has eight, 80-bit data registers, plus a status word and a control word, both 16 bits long. The status word is similar to the 8088's flags register. The control word governs the way the 8087 handles rounding, infinity, and precision.

Working with the 8087's data registers is quite different than working with the 8088's registers, in that you generally don't address them individually. Instead, you use them as a *stack*.

The 8087's Stack

You will recall that the 8088 uses a stack to hold return addresses during procedure calls and interrupt servicing, and that we can use it to preserve the contents of registers. A stack operates like a plate dispenser in a cafeteria; the last item you put on it is the first one you can remove from it. The 8087's data registers work in the same way. That is, you always put numbers into the top register, and when you do, it pushes the rest of the registers "down" one position in the stack. (Like the 8088, the 8087 does not actually move the registers, it simply changes the contents of a "stack pointer.")

Because the data registers operate as a stack, most of the 8087's instructions use the stack contents implicitly. For example, when you give the 8087 an add instruction, it adds the contents of two numbers on the stack and leaves the result on the stack. If you have worked with a stack-oriented language such as FORTH, you will feel quite comfortable programming the 8087; otherwise, it will probably take you some time to get used to this concept.

Floating-Point Format

The data registers hold numbers in "floating-point" format, the computer's version of scientific notation. In this notation, you would write −150 as

$$-1.5 \times 10^2$$

The 8087's data registers represent such numbers by splitting them into three fields: a 1-bit sign, a 15-bit exponent, and a 64-bit significand (or *man-*

tissa). Hence, a data register stores −150 with 1 in the sign field, a number representing 2 in the exponent field, and 15 in the significand field.

Fortunately, we can usually forget about how the 8087 stores numbers. But we *do* need to know about what kinds of data it can operate on.

12.2 Data Types

The 8087 can operate on seven types of data: three types of integers (word, short, and long), three types of reals (short, long, and temporary), and packed decimal. Table 12-1 summarizes the data types.

Of the integer types, the two-byte *word integer*, which corresponds to BASIC's integer data type, is useful for indexing arrays and other data structures. Since a word integer can hold values from −32,768 to 32,767, you probably won't find many uses for the short and long integer types.

Short real and *long real* correspond to BASIC's single- and double-precision data types, respectively. Short real numbers are accurate to about seven decimal places. This means that numbers that differ in only the eighth place will look the same to the 8087. (For example, it treats 1.23456789 the same as 1.12345681.) Hence, short real numbers are convenient for storing input data, but you should perform calculations using long real numbers, to minimize the effect of round-offs. Long real numbers are accurate to about 16 places.

Table 12-1. 8087 data types.

Data Type	*Bits*	*Significant Digits*	*Range*
Word Integer	16	4 or 5	$-32{,}768$ to $32{,}767$
Short Integer	32	9	-2×10^{9} to 2×10^{9}
Long Integer	64	18	-9×10^{18} to 9×10^{18}
Short Real	32	6 or 7	10^{37} to 10^{38}
Long Real	64	15 or 16	10^{307} to 10^{308}
Temporary Real	80	19	10^{4932} to 10^{4932}
Packed Decimal	80	18	18 decimal digits + sign

Temporary real is the format the 8087 uses to store numbers in its data registers. Since this format uses a 64-bit significand, every other data type will fit into it without loss of precision. With its 80-bit length, the temporary real format shields the user from cumulative rounding errors and from overflow or underflow in intermediate calculations.

Finally, *packed decimal* is used for business and data processing. It can hold up to 18 significant digits, and "packs" them two per byte in memory. You will recall this approach from our discussion of binary-coded decimal (BCD) numbers in Chapter 3.

12.3 Instruction Set

We can divide the 8087's instruction set into six groups: data transfer, arithmetic, comparison, transcendental, constants, and control. Table 12-2 gives their general forms (as usual, square brackets mark optional items) and a short description.

The *data transfer* instructions move numbers to and from the top of the 8087's data register stack.

Table 12-2. 8087 instruction set.

Instruction	*Description*
Data Transfer	
FBLD source	Load packed decimal
FBSTP dest	Store and pop packed decimal
FILD source	Load integer
FIST dest	Store integer
FISTP dest	Store and pop integer
FLD source	Load real
FST dest	Store real
FSTP dest	Store real and pop
FXCH [dest]	Exchange registers
Arithmetic	
FADD [dest,[source]]	Add real
FADDP dest,source	Add real and pop
FIADD source	Add integer
FSUB [dest,[source]]	Subtract real
FSUBP dest,source	Subtract real and pop
FISUB source	Subtract integer
FSUBR [dest,[source]]	Subtract real reversed
FSUBRP dest,source	Subtract real reversed and pop
FISUBR source	Subtract integer reversed
FMUL [dest,[source]]	Multiply real
FMULP dest,source	Multiply real and pop
FIMUL source	Multiply integer
FDIV [dest,[source]]	Divide real
FDIVP dest,source	Divide real and pop
FIDIV source	Divide integer
FDIVR [dest,[source]]	Divide real reversed
FDIVRP dest,source	Divide real reversed and pop
FIDIVR source	Divide integer reversed
FSQRT	Square root

Table 12-2. 8087 instruction set (continued).

Instruction	*Description*
FSCALE	Scale by a power of two
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign
Comparison	
FCOM [source]	Compare real
FCOMP [source]	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM source	Compare integer
FICOMP source	Compare integer and pop
FTST	Test top of stack for zero
FXAM	Examine top of stack
Transcendental	
F2XM1	2 to the X power, minus 1
FYL2X	Y times $\log_2 X$
FYL2XP1	Y times $\log_2(X+1)$
FPTAN	Partial tangent
FPATAN	Partial arctangent
Constants	
FLDZ	Load 0.0
FLD1	Load 1.0
FLDPI	Load pi
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load common logarithm of 2
FLDLN2	Load natural logarithm of 2
Control	
FLDCW source	Load control word
FSTCW dest	Store control word
FSTSW dest	Store status word
FSAVE dest	Save state
FRSTOR source	Restore state
FLDENV source	Load environment
FSTENV dest	Store environment
FWAIT	Wait (halt the 8088)

Table 12-2. 8087 instruction set (continued).

Instruction	*Description*
FINIT	Initialize (reset) the 8087
FENI	Enable interrupts
FDISI	Disable interrupts
FCLEX	Clear exceptions
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE dest	Free (clear) register
FNOP	No operation

The *arithmetic* instructions provide the four basic operations (add, subtract, multiply, and divide), plus other convenient functions such as square root and absolute value. The subtract and divide instructions come in two forms. With the standard forms, you subtract a source from a destination or divide a destination by a source. With the "reversed" forms, you subtract a destination from a source or divide a source by a destination. The reversed forms let you leave results in memory, which may only be a source operand.

The *comparison* instructions compare the top number on the stack to another stack number or a memory location. Compares are convenient for finding the largest number in an array or determining whether a number is less than, equal to, or greater than zero.

The *transcendental* instructions compute logarithms and trigonometric functions. Although the 8087 provides only tangent and arctangent instructions, it is fairly easy to derive other trigonometric functions from them.

The *constant* instructions push any of seven constants onto the stack: zero, one, pi, or four logarithmic values. In all cases, the 8087 gives them full temporary real (19-digit) accuracy.

The *control* instructions let you preserve status information, change the way the 8087 rounds results, enable and disable interrupts, and do a variety of other "housekeeping" jobs. One instruction, FWAIT, generates an 8088 WAIT instruction, which prevents it from accessing a memory location that the 8087 is using.

12.4 8087 Programming with the Macro Assembler

You can develop 8087 programs just like 8088 programs, except you must tell the assembler that you are using 8087 instructions. You can do this in either of two ways:

- Enter the command .8087 at the very beginning of the main program module—just after the TITLE statement.
- Assemble the program using the form

  masm *filename*/**R**

Constants

In Section 2.3 we described the four kinds of constants the assembler accepts for regular 8088 programming; these are binary, decimal, hexadecimal, and character. In your work with the 8087, you may use two additional forms:

1. *Decimal real*—A number with a decimal point; for example, 3.14159.
2. *Decimal scientific*—A decimal number followed by E and an exponent value; for example, 2.654E−12.

Data Definition Pseudo-Ops

In Section 12.2, we described the seven types of data the 8087 can accept: three integers (word, short, and long), three reals (short, long, and temporary), and packed decimal. The MACRO Assembler provides data definition pseudo-ops that allow us to set up memory variables for each type, as follows:

- For word integers, which are 16 bits long, use *DW* (Define Word).
- For short integers and short reals, which are 32 bits long, use *DD* (Define Doubleword).
- For long integers and long reals, which are 64 bits long, use *DQ* (Define Quadword).
- For temporary reals and packed integers, which are 80 bits long, use *DT* (Define Tenbytes).

Note that DQ and DT are new.

Examples of each form are:

```
WORD_INTEGER    DW  16483
SHORT_INTEGER   DD  145897
LONG_INTEGER    DQ  42765844
SHORT_REAL      DD  3.14159
LONG_REAL       DQ  1.0E-14
TEMP_REAL       DT  1.5E-16
PACKED_DEC      DT  -1457594442553
```

12.5 Summary

The 8087 Math Coprocessor is a powerful device that can dramatically improve your computer's ability to perform mathematical calculations. It accepts seven different types of data—three integer, three real, and packed decimal—giving maximum flexibility for any kind of calculation you need. No matter what data type you use, the 8087 performs all internal calculations using an 80-bit floating-point format, which gives accuracy to 19 significant places. This not only means that your answers will be highly precise, but also that you will rarely encounter overflow or underflow errors.

The 8087 provides instructions for adding, subtracting, multiplying, and dividing both integer and real numbers. It also has a variety of logarithmic and trigonometric instructions, plus special ones that perform tasks you frequently need, such as extracting a square root and finding an absolute value. The slowest instruction takes 190 microseconds to execute and the fastest takes just one microsecond, but most require from 18 to 40 microseconds.

Programming the 8087 in assembly language may take some getting used to, because it has a stack-oriented architecture. However, the short time it takes you to accustom yourself more than makes up for the agony you would have to endure in writing multi-precision arithmetic procedures.

Answers to Study Exercises

Chapter 0. A Crash Course in Computer Numbering Systems

1. (a) 1100 (b) 10001 (c) 101101 (d) 1001000
2. (a) 8 (b) 21 (c) 31
3. (a) 8 (b) 15 (c) 1F
4. The hexadecimal value D8 can represent the unsigned number 216 or the signed number -40. To obtain the signed number, convert the hexadecimal value to binary (11011000), then reverse each bit (to make 00100111) and add 1 (to make 00101000, or decimal 40).

Chapter 1. Introduction to Assembly Language Programming

1. There is no difference between the instruction set of the 8088 and that of the 8086—they are identical.
2. When calculating a physical address, the 8088 automatically appends four zeros to the segment number to form the *segment address*. Therefore, it uses 4000H as 40000H and calculates the physical address as

   Physical address = 2H + 40000H = 40002H

3. If the AX register contains 1A2BH, AL (its low-order byte) contains 2BH.
4. Variables are usually stored in the data segment, so the Data Segment (DS) register is used to access them.
5. Bit 7, the Sign Flag (SF), is set to 1 if a subtraction gives a negative result.

Chapter 2. Using an Assembler

1. The assembler allocates one byte for VAR1, ten bytes (five words = ten bytes) for VAR2, and ten bytes for VAR3, for a total of *21* bytes.
2. The assembler doesn't put anything into VAR1. The ? operand tells it to simply *reserve* one byte for VAR1. Your program must put a value into VAR1.

   Note that the assembler only puts a value into one location here: it puts 20 into the fifth word of VAR2. It leaves all other locations "undefined."
3. The = statement can be redefined later in the program; the EQU statement cannot.
4. CONST is a *byte* variable, and cannot hold values greater than 255.
5. Every procedure must start with PROC and end with ENDP.
6. You can only call a NEAR procedure from within the segment in which it is defined. You can call a FAR procedure from any segment in the program.
7. The first procedure must be FAR because the calling program (DOS or DEBUG) is in a different segment.
8. This ASSUME tells the assembler to use CS to address any instruction labels in the CSEG segment. In other words, it identifies CSEG as a *code* segment rather than a data, stack, or extra segment.
9. The Linker generates a *relocatable run module* that DOS can store at any convenient place in memory. This frees you from having to decide where to store a program in memory. The Linker also combines two or more object modules.

Chapter 3. The 8088 Instruction Set

1. The instruction is MOV ES:SAVE_AX,AX.
2. This sequence stores 0 into the first location of the data segment (addressed by BX) and the stack segment (addressed by BP).
3. a. *Invalid.* A constant cannot be a destination.
   b. *Valid,* but since TEMP has not been initialized, you will get "garbage" in AL.
   c. *Invalid.* You can't move a word value into a byte variable.
   d. *Invalid.* The MOV instruction cannot be used to make a direct memory-to-memory transfer.
   e. *Invalid.* The assembler does not recognize the addressing form [BX][BP]. See Table 3-1 for valid operand formats.
4. These instructions clear the AX register:

```
SUB   AX,AX
MOV   AX,0
```

5. These instructions are identical. Both load the offset of location TABLE+4 into BX. However, LEA is both shorter and more explicit.
6. This loop subtracts V2 from V1:

```
        MOV   CX,3          ;Word count = 3
        MOV   BX,0          ;Offset = 0
        CLC                 ;Clear Carry (CF)
NEXT:   MOV   DX,V2[BX]     ;Subtract words
        SBB   V1[BX],DX
        INC   BX            ; and address next word
        INC   BX
        LOOP  NEXT
```

   Note that BX is increased by 2 after each subtraction because words lie two bytes apart in memory. We use two INCs rather than one ADD so as not to disturb the Carry Flag (CF), which the SBB operation uses.
7. This MUL instruction generates an error; you cannot multiply by an immediate value.
8. The results are:
   a. (AX) = 0220H
   b. (AX) = 5335H
   c. (AX) = 5115H
   d. (AX) = 0EDCBH
   e. (AX) = 1234H (Because TEST affects only the flags)
9. The sequence to normalize AX is:

```
        TEST  AX,0FFFFH
        JZ    NORM          ;Exit if (AX) = 0
        MOV   CX,15         ;Get set for 15 shifts
NEXT_BIT: JS  NORM          ;Exit if Bit 15 = 1
        SHL   AX,1          ;Otherwise shift AX left by one
        LOOP  NEXT_BIT
NORM:   ..
        ..
```

10. If you said the sequence subtracts 30 from AX, look again. LOOP decrements CX, then jumps to START if CX is zero. However, the MOV instruction continually reinitializes CX to 3, so CX will *never* reach zero. This kind of endless loop is a common programming error. Watch out for it.

Chapter 4. High-Precision Mathematics

1. This procedure extracts a square root by subtracting successively higher odd numbers:

```
SR32    PROC
        PUSH  AX        ;Save original number
        PUSH  DX
        PUSH  CX        ; and CX on the stack
        MOV   BX,1      ;To start, (BX) = 1
        SUB   CX,CX     ; and (CX) = 0
AGAIN:  SUB   AX,BX     ;Subtract next odd number from AX
        SBB   DX,0      ; and DX
        JC    DONE      ;Did this subtraction create a borrow?
        INC   CX        ; No.  Increase the square root by 1,
        ADD   BX,2      ;  calculate the next odd number,
        JMP   AGAIN     ;  then go make the next subtraction
DONE:   MOV   BX,CX     ; Yes.  Transfer result to BX,
        POP   CX        ;  and restore the registers
        POP   DX
        POP   AX
        RET
SR32    ENDP
```

Chapter 5. Operating on Data Structures

1. This modified version of Example 5-1 can be used to build a list from scratch, as well as to add a new element to an existing list:

```
ADD_TO_UL  PROC
           PUSH  DI              ;Save starting address
           MOV   CX,ES:[DI]      ;Fetch word count
           ADD   DI,2            ;Make DI point to 1st element
           CMP   CX,0            ;Is the list null?
           JE    ADD_IT          ; Yes.  Value is 1st element
           CLD                   ; No.  Set DF = 0
REPNE      SCASW                 ;Value already in the list?
           JNE   ADD_IT
           POP   DI              ; Yes.  Restore starting addr.
           RET                   ;  and exit
ADD_IT:    STOSW                 ; No.  Add it to end of list
           POP   DI              ;  then update element count
           INC   WORD PTR ES:[DI]
           RET
ADD_TO_UL  ENDP
```

2. Here is the find-and-replace procedure:

```
;  This procedure searches an ordered list in the extra
;  segment for the word value contained in AX.  If a
;  matching element is found, its contents are replaced
```

```
;  with the value in BX.
;  AX and BX are unaffected.
;
REPLACE  PROC
         CALL   B_SEARCH    ;Is the search value in the list?
         JC     QUIT        ; If not, exit
         MOV    ES:[SI],BX  ; Is so, replace it with BX
QUIT:    RET
REPLACE  ENDP
```

Chapter 6. Using the System Resources

1. The following procedure measures the elapsed time between two key depressions, and returns the hours in CH (the operator took a coffee break between key depressions), the minutes in CL, the seconds in DH, and 1/100 seconds in DL. Other registers are unaffected.

```
TIME_KEYS  PROC
           PUSH   AX
           MOV    AH,7    ;When the first key is pressed,
           INT    21H
           SUB    CX,CX   ; set time count to zero
           SUB    DX,DX
           MOV    AH,2DH
           INT    21H
           MOV    AH,7    ;When another key is pressed,
           INT    21H
           MOV    AH,2CH  ; read the count
           INT    21H
           POP    AX      ;Restore AX
           RET            ; and exit
TIME_KEYS  ENDP
```

2. The author got a minimum count of DX = 10H, or 0.16 seconds. However, years of wine, women, song, and computer programming have taken their toll; you may do better. At any rate, this test shows how much faster computers are than humans. The processor can execute *thousands* of instructions in the same time it takes you to press a key twice!
3. I probably entered my messages with statements such as

```
MSG1   DB   'Sort Program$'
MSG2   DB   'Press any key to begin.$'
```

To put them on separate lines, include Carriage Return and Line Feed characters in the string, as follows:

```
MSG1   DB   'Sort Program',ODH,OAH,'$'
MSG2   DB   'Press any key to begin.',ODH,OAH,'$'
```

4. To display the number in CX, you must convert it to ASCII by adding 30H. The program you want consists of a data segment with the following statements:

```
MSG_START   DB   'Try again.  You have $'
MSG_END     DB   ' starfighters left.',ODH,OAH,'$'
```

and a code segment that includes

```
LEA   DX,MSG_START   ;Display start of message
MOV   AH,9
INT   21H
MOV   DL,CL          ;Convert number to ASCII
ADD   DL,3OH
MOV   AH,6           ; and display it
LEA   DX,MSG_END     ;Display rest of message
MOV   AH,9
INT   21H
```

Chapter 7. Graphics Made Easy

1. This procedure moves a "bird" across the screen, on line 20:

```
            EXTRN   DELAY:FAR
MOVE_BIRD   PROC
            PUSH    AX          ;Save registers
            PUSH    BX
            PUSH    CX
            PUSH    DX
            MOV     AH,15       ;Set BH to active display page
            INT     1OH
            MOV     AH,O        ;Select 80x25, B/W, Alphanumeric
            INT     1OH
            MOV     CX,1        ;Character count = 1
            MOV     DH,2O       ;Start at row 20,
            MOV     DL,O        ; column O
DSPLY_V:    MOV     BL,'v'      ;Display a "v"
            CALL    NEXT_BIRD
            CMP     DL,8O       ;Done?
            JE      BIRD_DONE
            MOV     BL,OC4H     ; No.  Display a "dash"
            CALL    NEXT_BIRD
            CMP     DL,8O       ;Done?
            JNE     DSPLY_V     ; If not, begin again
```

```
BIRD_DONE: PUSH   DX              ; Otherwise, restore registers
           PUSH   CX
           PUSH   BX
           PUSH   AX
           RET                    ;  and exit
MOVE_BIRD  ENDP
;
;  This procedure displays the bird (either "v" or "dash")
;  for 1/10 second, then erases it and advances DL to
;  the next column.
;
NEXT_BIRD   PROC
            MOV    AH,2        ;Move cursor to next position
            INT    10H
            MOV    AH,10       ;Write character to the screen
            MOV    AL,BL
            INT    10H
            CALL   DLY_TENTH   ;Wait 1/10-second
            SUB    AL,AL       ; then erase the bird
            MOV    AH,10
            INT    10H
            INC    DL          ;Point to next column
            RET                ; and return to caller
NEXT_BIRD    ENDP
;
;  This procedure generates a 1/10-second delay.
;
DLY_TENTH    PROC
            SUB    AL,AL       ;Set minutes to zero
            SUB    BH,BH       ;Set seconds to zero
            MOV    BL,10       ;Set hundredths to 10
            CALL   DELAY
            RET
DLY_TENTH    ENDP
```

Admittedly, the shape this program produces doesn't look much like a bird, but that's the price we pay for using block graphics rather than dot graphics. If you come up with a better program that moves and changes a shape, the author would like to hear about it.

2. The following procedure, which moves a smile face randomly, is a combination of Examples 6-1 and 7-2. The face is displayed for a quarter-second between moves.

```
            EXTRN  DELAY:FAR
RAND_FACE   PROC
            PUSH   AX          ;Save registers
```

```
            PUSH    BX
            PUSH    CX
            PUSH    DX
            MOV     AH,15       ;Set BH to active display page
            INT     10H
            MOV     AH,0        ;Select 80x25, B/W, Alphanumeric
            INT     10H
            MOV     CX,1        ;Character count = 1
INIT_LINE:  CALL    RAND_8      ;Get a random line number
            AND     DL,1FH
            CMP     DL,24       ; between 0 and 24,
            JA      INIT_LINE
            MOV     DH,DL       ; and put it in DH
            MOV     DL,0        ;Start at column 0
NEXT_FACE:  MOV     AH,2        ;Move cursor to next position
            INT     10H
            MOV     AL,2        ;Display smile face
            MOV     AH,10
            INT     10H
            CALL    DLY_QRTR    ;Wait a quarter-second
            SUB     AL,AL       ; then erase the face
            MOV     AH,10
            INT     10H
            MOV     AX,DX       ;Save row,column in AX
LINE_IND:   CALL    RAND_8      ;Get a number between 0 and 2
            AND     DL,3
            CMP     DL,2
            JNE     CHK_3
            MOV     DX,AX       ;It's a 2.  Do nothing
            JE      INC_COL
CHK_3:      JA      LINE_IND
            CMP     DL,1
            JNE     ITS_0
            MOV     DX,AX       ;It's a 1.  Increment row
            INC     DH
            CMP     DH,25       ; and make sure row < 25
            JE      EXIT_FACE
            JNE     INC_COL
ITS_0:      MOV     DX,AX       ;It's a 0.  Decrement row
            DEC     DH
            JS      EXIT_FACE   ; and make sure row is positive
INC_COL:    INC     DL          ;Point to next column
            CMP     DL,80       ; and exit if it's 80
            JNE     NEXT_FACE
EXIT_FACE:  POP     DX          ;Restore registers
```

```
            POP     CX
            POP     BX
            POP     AX
            RET                     ; and exit
RAND_FACE   ENDP
;
;  This procedure generates a quarter-second delay.
;
DLY_QRTR    PROC
            SUB     AL,AL           ;Set minutes to zero
            SUB     BH,BH           ;Set seconds to zero
            MOV     BL,25           ;Set hundredths to 25
            CALL    DELAY
            RET
DLY_QRTR    ENDP
;
;  This procedure returns a 16-bit random number in DX, but
;  the caller, RAND_FACE, only uses the value in DL.
;
RAND_8      PROC
            PUSH    AX              ;Save AX and CX
            PUSH    CX
            MOV     AH,0            ;Read the timer
            INT     1AH
            POP     CX              ; and exit
            POP     AX
            RET
RAND_8      ENDP
```

Hexadecimal/Decimal Conversion *A*

HEXADECIMAL COLUMNS											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Byte				Byte				Byte			

POWERS OF 2

2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

$2^0 = 16^0$
$2^4 = 16^1$
$2^8 = 16^2$
$2^{12} = 16^3$
$2^{16} = 16^4$
$2^{20} = 16^5$
$2^{24} = 16^6$
$2^{28} = 16^7$
$2^{32} = 16^8$
$2^{36} = 16^9$
$2^{40} = 16^{10}$
$2^{44} = 16^{11}$
$2^{48} = 16^{12}$
$2^{52} = 16^{13}$
$2^{56} = 16^{14}$
$2^{60} = 16^{15}$

POWERS OF 16

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

B

IBM PC ASCII Character Set

HIGH PART

LOW PART

DECIMAL VALUE	➡	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
⬇	HEXA DECIMAL VALUE	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	BLANK (NULL)	►	BLANK (SPACE)	0	@	P	‘	p	Ç	É	á	1/4 Dots On	┐	╨	∝	≡
1	1	☺	◄	!	1	A	Q	a	q	ü	Æ	í	1,2 Dots On	┘	╤	β	±
2	2	☻	↕	"	2	B	R	b	r	é	FE	ó	3,4 Dots On	┌	╥	γ	≥
3	3	♥	!!	#	3	C	S	c	s	â	ô	ú		█	╙	π	≤
4	4	♦	¶	$	4	D	T	d	t	ä	ö	ñ		▄	╘	Σ	⌠
5	5	♣	§	%	5	E	U	e	u	à	ò	Ñ		▌	╒	σ	⌡
6	6	♠	▬	&	6	F	V	f	v	å	û	ª		▐	╓	μ	÷
7	7	BEEP •	↨	'	7	G	W	g	w	ç	ù	º		▀	╫	τ	≈
8	8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿		α	╪	Φ	°
9	9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐		ß	┘	Θ	•
10	A	◙	→	*	:	J	Z	j	z	è	Ü	¬		Γ	┌	Ω	·
11	B	♂	ESC ←	+	;	K	[	k	{	ï	¢	½		π	╫	δ	√
12	C	♀	∟	,	<	L	\	l	¦	î	£	¼		Σ	╪	∞	η
13	D	♪	↔	—	=	M	]	m	}	ì	¥	¡		σ	┘	∅	2
14	E	♫	▲	.	>	N	^	n	~	Ä	Pts	«		µ	┌	∈	■
15	F	☼	▼	/	?	O	_	o	△	Å	f	»		τ	╫	∩	BLANK 'FF'

C

8088 Instruction Times

You can use the two tables in this appendix to calculate how long the PC's 8088 microprocessor will take to execute instructions in your programs. These tables show execution times in *clock cycles*. To convert them to nanoseconds, multiply by 210.

Table C-2 lists the number of *Clocks* each instruction takes to execute and the number of *Bytes* it occupies in memory. When using Table C-2, note the following:

1. Some instructions take more time to execute if their operands are word values rather than byte values. For these instructions, execution times are listed in the form *b(w)*, where *b* and *w* indicate the number of clock cycles for byte and word operands, respectively.
2. Most instructions that reference memory include the abbreviation EA in the Clocks column. This tells you that additional clock cycles are required to calculate the Effective Address; these incremental times are listed in Table C-1.
3. The execution time of conditional jump instructions and loop instructions depends on whether the transfer is made. If the transfer *is* made, use the larger number in the Clocks column. Otherwise, if execution "drops through" to the next instruction, use the smaller number.

For example, suppose you want to find the execution time of the instruction

```
ADD  ES:[BX],DX
```

This instruction has the general form *ADD memory,register*, so its execution time is given by the equation

$$9(24) + EA$$

Since this particular ADD instruction operates on word values, you must use the 24 instead of the 9. Because the equation contains EA, you must add an address calculation time from Table C-1. The destination operand has the form [BX], which tells you to add five additional clock cycles. However, because this operand has a segment override (*ES:*), you must add two more clock cycles. Combining these three values gives a total execution time of 31 clock cycles (24+5+2).

Table C-l. Effective address calculation time.

EA Components	*Operand Formats*	*Clocks**
Displacement Only	disp label	6
Base or Index Only	[BX] [BP] [DI] [SI]	5
Displacement + Base or Index	[BX]+disp [BP]+disp [DI]+disp [SI]+disp	9
Base + Index	[BX][SI] [BX][DI]	7
	[BP][SI] [BP][DI]	8
Displacement + Base + Index	[BX][SI]+disp [BX][DI]+disp	11
	[BP][SI]+disp [BP][DI]+disp	12

*Add two clocks for segment override.

Table C-2. Instruction times.

Instruction	*Clocks*	*Bytes*
AAA	4	1
AAD	60	2
AAM	83	1
AAS	4	1
ADC register,register	3	2
ADC register,memory	9(13)+EA	2-4

Table C-2. Instruction times (continued).

Instruction	*Clocks*	*Bytes*
ADC memory,register	16(24)+EA	2-4
ADC register,immediate	4	3-4
ADC memory,immediate	17(25)+EA	3-6
ADC accumulator,immediate	4	2-3
ADD register,register	3	2
ADD register,memory	9(13)+EA	2-4
ADD memory,register	16(24)+EA	2-4
ADD register,immediate	4	3-4
ADD memory,immediate	17(25)+EA	3-6
ADD accumulator,immediate	4	2-3
AND register,register	3	2
AND register,memory	9(13)+EA	2-4
AND memory,register	16(24)+EA	2-4
AND register,immediate	4	3-6
AND memory,immediate	17(15)+EA	3-6
AND accumulator,immediate	4	2-3
CALL near-proc	23	3
CALL far-proc	36	5
CALL memptr16	29	2-4
CALL regptr16	24	2
CALL memptr32	57	2-4
CBW	2	1
CLC	2	1
CLD	2	1
CLI	2	1
CMC	2	1
CMP register,register	3	2
CMP register,memory	9(13)+EA	2-4
CMP memory,register	9(13)+EA	2-4
CMP register,immediate	4	3-4
CMP memory,immediate	10(14)+EA	3-6
CMP accumulator,immediate	4	2-3
CMPS dest-string, source-string	22(30)	1
CMPS (repeat) dest-string, source-string	9+22(30)/rep	1
CWD	5	1
DAA	4	1
DAS	4	1
DEC reg16	2	1

Table C-2. Instruction times (continued).

Instruction	*Clocks*	*Bytes*
DEC reg8	3	2
DEC memory	15(23) + EA	2-4
DIV reg8	80-90	2
DIV reg16	144-162	2
DIV mem8	(86-96) + EA	2-4
DIV mem16	(154-172) + EA	2-4
ESC immediate,memory	8(35) + EA	2-4
ESC immediate,register	2	2
HLT	2	1
IDIV reg8	101-112	2
IDIV reg16	165-184	2
IDIV mem8	(107-118) + EA	2-4
IDIV mem16	(175-194) + EA	2-4
IMUL reg8	80-98	2
IMUL reg16	128-154	2
IMUL mem8	(86-104) + EA	2-4
IMUL mem16	(138-164) + EA	2-4
IN accumulator,immed8	10(14)	2
IN accumulator,DX	8(12)	1
INC reg16	2	1
INC reg8	3	2
INC memory	15(23) + EA	2-4
INT 3	52	1
INT immed8 (not type 3)	51	2
INTO	53 or 4	1
IRET	32	1
All conditional jump instructions except JCXZ:		
Jccc short-label	16 or 4	2
JCXZ short-label	18 or 6	2
JMP short-label	15	2
JMP near-label	15	3
JMP far-label	15	5
JMP memptr16	18 + EA	2-4
JMP regptr16	11	2
JMP memptr32	24 + EA	2-4
LAHF	4	1
LDS reg16,mem32	24 + EA	2-4
LEA reg16,mem16	2 + EA	2-4
LES reg16,mem32	24 + EA	2-4

Table C-2. Instruction times (continued).

Instruction	*Clocks*	*Bytes*
LOCK	2	1
LODS source-string	12(16)	1
LODS (repeat) source-string	9+13(17)/rep	1
LOOP short-label	17 or 5	2
LOOPE/LOOPZ short-label	18 or 6	2
LOOPNE/LOOPNZ short-label	19 or 5	2
MOV memory,accumulator	10(14)	3
MOV accumulator,memory	10(14)	3
MOV register,register	2	2
MOV register,memory	8(12)+EA	2-4
MOV memory,register	9(13)+EA	2-4
MOV register,immediate	4	2-3
MOV memory,immediate	10(14)+EA	3-6
MOV seg-reg,reg16	2	2
MOV seg-reg,mem16	8(12)+EA	2-4
MOV reg16,seg-reg	2	2
MOV memory,seg-reg	9(13)+EA	2-4
MOVS dest-string, source-string	18(26)	1
MOVS (repeat) dest-string, source-string	9+17(25)/rep	1
MUL reg8	70-77	2
MUL reg16	118-133	2
MUL mem8	(76-83)+EA	2-4
MUL mem16	(128-143)+EA	2-4
NEG register	3	2
NEG memory	16(24)+EA	2-4
NOP	3	1
NOT register	3	2
NOT memory	16(24)+EA	2-4
OR register,register	3	2
OR register,memory	9(13)+EA	2-4
OR memory,register	16(24)+EA	2-4
OR register,immediate	4	3-6
OR memory,immediate	17(15)+EA	3-6
OR accumulator,immediate	4	2-3
OUT immed8,accumulator	10(14)	2
OUT DX,accumulator	8(12)	1
POP register	12	1
POP seg-reg (CS illegal)	12	1
POP memory	25+EA	2-4

Table C-2. Instruction times (continued).

Instruction	*Clocks*	*Bytes*
POPF	12	1
PUSH register	15	1
PUSH seg-reg (CS legal)	14	1
PUSH memory	24+EA	2-4
PUSHF	14	1
RCL register,1	2	2
RCL register,CL	8+4/bit	2
RCL memory,1	15(23)+EA	2-4
RCL memory,CL	20(28)+EA+4/bit	2-4
RCR register,1	2	2
RCR register,CL	8+4/bit	2
RCR memory,1	15(23)+EA	2-4
RCR memory,CL	20(28)+EA+4/bit	2-4
REP	2	1
REPE/REPZ	2	1
REPNE/REPNZ	2	1
RET (intra-seg, no pop)	20	1
RET (intra-seg, pop)	24	3
RET (inter-seg, pop)	32	1
RET (inter-seg, no pop)	31	3
ROL register,1	2	2
ROL register,CL	8+4/bit	2
ROL memory,1	15(23)+EA	2-4
ROL memory,CL	20(28)+EA+4/bit	2-4
ROR register,1	2	2
ROR register,CL	8+4/bit	2
ROR memory,1	15(23)+EA	2-4
ROR memory,CL	20(28)+EA+4/bit	2-4
SAHF	4	1
SAL/SHL register,1	2	2
SAL/SHL register,CL	8+4/bit	2
SAL/SHL memory,1	15(23)+EA	2-4
SAL/SHL memory,CL	20(28)+EA+4/bit	2-4
SAR register,1	2	2
SAR register,CL	8+4/bit	2
SAR memory,1	15(23)+EA	2-4
SAR memory,CL	20(28)+EA+4/bit	2-4
SBB register,register	3	2
SBB register,memory	9(13)+EA	2-4
SBB memory,register	16(24)+EA	2-4
SBB register,immediate	4	3-4

Table C-2. Instruction times (continued).

Instruction	*Clocks*	*Bytes*
SBB memory,immediate	17(25)+EA	3-6
SBB accumulator,immediate	4	2-3
SCAS dest-string	15(19)	1
SCAS (repeat) dest-string	9+15(19)/rep	1
SHR register,1	2	2
SHR register,CL	8+4/bit	2
SHR memory,1	15(23)+EA	2-4
SHR memory,CL	20(28)+EA+4/bit	2-4
STC	2	1
STD	2	1
STI	2	1
STOS dest-string	11(15)	1
STOS (repeat) dest-string	9+10(14)/rep	1
SUB register,register	3	2
SUB register,memory	9(13)+EA	2-4
SUB memory,register	16(24)+EA	2-4
SUB register,immediate	4	3-4
SUB memory,immediate	17(25)+EA	3-6
SUB accumulator,immediate	4	2-3
TEST register,register	3	2
TEST register,memory	9(13)+EA	2-4
TEST register,immediate	5	3-4
TEST memory,immediate	11+EA	3-6
TEST accumulator,immediate	4	2-3
WAIT	3+5n	1
XCHG accumulator,reg16	3	1
XCHG memory,register	17(25)+EA	2-4
XCHG register,register	4	2
XLAT source-table	11	1
XOR register,register	3	2
XOR register,memory	9(13)+EA	2-4
XOR memory,register	16(24)+EA	2-4
XOR register,immediate	4	3-6
XOR memory,immediate	17(15)+EA	3-6
XOR accumulator,immediate	4	2-3

D

8088 Instruction Set Summary

Table D-1 summarizes the 8088 instruction set in alphabetical order. For each instruction, it shows the general assembler format and which flags are affected. In the *Flags* column, - means unchanged, * means may have changed, and ? means undefined.

Table D-1. 8088 instruction set.

			Flags								
Mnemonic	*Assembler Format*		*OF*	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
AAA	AAA		?	-	-	-	?	?	*	?	*
AAD	AAD		?	-	-	-	*	*	?	*	?
AAM	AAM		?	-	-	-	*	*	?	*	?
AAS	AAS		?	-	-	-	?	?	*	?	*
ADC	ADC	destination,source	*	-	-	-	*	*	*	*	*
ADD	ADD	destination,source	*	-	-	-	*	*	*	*	*
AND	AND	destination,source	0	-	-	-	*	*	?	*	0
CALL	CALL	target	-	-	-	-	-	-	-	-	-
CBW	CBW		-	-	-	-	-	-	-	-	-
CLC	CLC		-	-	-	-	-	-	-	-	0
CLD	CLD		-	0	-	-	-	-	-	-	-
CLI	CLI		-	-	0	-	-	-	-	-	-
CMC	CMC		-	-	-	-	-	-	-	-	*
CMP	CMP	destination,source	*	-	-	-	*	*	*	*	*
CMPS	CMPS	dest-string,source-string	*	-	-	-	*	*	*	*	*
CMPSB	CMPSB		*	-	-	-	*	*	*	*	*
CMPSW	CMPSW		*	-	-	-	*	*	*	*	*
CWD	CWD		-	-	-	-	-	-	-	-	-
DAA	DAA		?	-	-	-	*	*	*	*	*

Table D-1. 8088 instruction set (continued).

Mnemonic	*Assembler Format*		*Flags*								
			OF	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
DAS	DAS		?	-	-	-	*	*	*	*	*
DEC	DEC	destination	*	-	-	-	*	*	*	*	-
DIV	DIV	source	?	-	-	-	?	?	?	?	?
ESC	ESC	ext-opcode,source	-	-	-	-	-	-	-	-	-
HLT	HLT		-	-	-	-	-	-	-	-	-
IDIV	IDIV	source	?	-	-	-	?	?	?	?	?
IMUL	IMUL	source	*	-	-	-	?	?	?	?	*
IN	IN	accumulator,port	-	-	-	-	-	-	-	-	-
INC	INC	destination	*	-	-	-	*	*	*	*	-
INT	INT	interrupt-type	-	-	0	0	-	-	-	-	-
INTO	INTO		-	-	0	0	-	-	-	-	-
IRET	IRET		*	*	*	*	*	*	*	*	*
JA/JNBE	JA	short-label	-	-	-	-	-	-	-	-	-
JAE/JNB	JAE	short-label	-	-	-	-	-	-	-	-	-
JB/JNAE/JC	JB	short-label	-	-	-	-	-	-	-	-	-
JBE/JNA	JBE	short-label	-	-	-	-	-	-	-	-	-
JCXZ	JCXZ	short-label	-	-	-	-	-	-	-	-	-
JE/JZ	JE	short-label	-	-	-	-	-	-	-	-	-
JG/JNLE	JG	short-label	-	-	-	-	-	-	-	-	-
JGE/JNL	JGE	short-label	-	-	-	-	-	-	-	-	-
JL/JNGE	JL	short-label	-	-	-	-	-	-	-	-	-
JLE/JNG	JLE	short-label	-	-	-	-	-	-	-	-	-
JMP	JMP	target	-	-	-	-	-	-	-	-	-
JNC	JNC	short-label	-	-	-	-	-	-	-	-	-
JNE/JNZ	JNE	short-label	-	-	-	-	-	-	-	-	-
JNO	JNO	short-label	-	-	-	-	-	-	-	-	-
JNP/JPO	JNP	short-label	-	-	-	-	-	-	-	-	-
JNS	JNS	short-label	-	-	-	-	-	-	-	-	-
JO	JO	short-label	-	-	-	-	-	-	-	-	-
JP/JPE	JP	short-label	-	-	-	-	-	-	-	-	-
JS	JS	short-label	-	-	-	-	-	-	-	-	-
LAHF	LAHF		-	-	-	-	-	-	-	-	-
LDS	LDS	reg16,mem32	-	-	-	-	-	-	-	-	-
LEA	LEA	reg16,mem16	-	-	-	-	-	-	-	-	-
LES	LES	reg16,mem32	-	-	-	-	-	-	-	-	-
LOCK	LOCK		-	-	-	-	-	-	-	-	-
LODS	LODS	source-string	-	-	-	-	-	-	-	-	-
LODSB	LODSB		-	-	-	-	-	-	-	-	-
LODSW	LODSW		-	-	-	-	-	-	-	-	-
LOOP	LOOP	short-label	-	-	-	-	-	-	-	-	-
LOOPE/ LOOPZ	LOOPE	short-label	-	-	-	-	-	-	-	-	-

Table D-1. 8088 instruction set (continued).

Mnemonic	*Assembler Format*		*Flags* OF	DF	IF	TF	SF	ZF	AF	PF	CF
LOOPNE/ LOOPNZ	LOOPNE	short-label	-	-	-	-	-	-	-	-	-
MOV	MOV	destination,source	-	-	-	-	-	-	-	-	-
MOVS	MOVS	dest-string,source-string	-	-	-	-	-	-	-	-	-
MOVSB	MOVSB		-	-	-	-	-	-	-	-	-
MOVSW	MOVSW		-	-	-	-	-	-	-	-	-
MUL	MUL	source	*	-	-	-	?	?	?	?	*
NEG	NEG	destination	*	-	-	-	*	*	*	*	*
NOP	NOP		-	-	-	-	-	-	-	-	-
NOT	NOT	destination	-	-	-	-	-	-	-	-	-
OR	OR	destination,source	0	-	-	-	*	*	?	*	0
OUT	OUT	port,accumulator	-	-	-	-	-	-	-	-	-
POP	POP	destination	-	-	-	-	-	-	-	-	-
POPF	POPF		*	*	*	*	*	*	*	*	*
PUSH	PUSH	source	-	-	-	-	-	-	-	-	-
PUSHF	PUSHF		-	-	-	-	-	-	-	-	-
RCL	RCL	destination,count	*	-	-	-	-	-	-	-	*
RCR	RCR	destination,count	*	-	-	-	-	-	-	-	*
REP	REP		-	-	-	-	-	-	-	-	-
REPE/REPZ	REPE		-	-	-	-	-	-	-	-	-
REPNE/ REPNZ	REPNE		-	-	-	-	-	-	-	-	-
RET	[pop-value]		-	-	-	-	-	-	-	-	-
ROL	ROL	destination,count	*	-	-	-	-	-	-	-	*
ROR	ROR	destination,count	*	-	-	-	-	-	-	-	*
SAHF	SAHF		-	-	-	-	*	*	*	*	*
SAL/SHL	SAL	destination,count	*	-	-	-	*	*	?	*	*
SAR	SAR	destination,count	*	-	-	-	*	*	?	*	*
SBB	SBB	destination,source	*	-	-	-	*	*	*	*	*
SCAS	SCAS	dest-string	*	-	-	-	*	*	*	*	*
SCASB	SCASB		*	-	-	-	*	*	*	*	*
SCASW	SCASW		*	-	-	-	*	*	*	*	*
SHR	SHR	destination,count	*	-	-	-	0	*	?	*	*
STC	STC		-	-	-	-	-	-	-	-	1
STD	STD		-	1	-	-	-	-	-	-	-
STI	STI		-	-	1	-	-	-	-	-	-
STOS	STOS	dest-string	-	-	-	-	-	-	-	-	-
STOSB	STOSB		-	-	-	-	-	-	-	-	-
STOSW	STOSW		-	-	-	-	-	-	-	-	-
SUB	SUB	destination,source	*	-	-	-	*	*	*	*	*

Table D-1. 8088 instruction set (continued).

Mnemonic	*Assembler Format*		*Flags*								
			OF	*DF*	*IF*	*TF*	*SF*	*ZF*	*AF*	*PF*	*CF*
TEST	TEST	destination,source	0	-	-	-	*	*	?	*	0
WAIT	WAIT		-	-	-	-	-	-	-	-	-
XCHG	XCHG	destination,source	-	-	-	-	-	-	-	-	-
XLAT	XLAT	source-table	-	-	-	-	-	-	-	-	-
XOR	XOR	destination,source	0	-	-	-	*	*	?	*	0

E

Disk User's Guide

The optional disk holds over 40 programs from this book. It is a reversible or "flippy" disk, so you must physically turn the disk over to use the programs on the back. The front side contains *source programs,* including the models we presented in Chapter 2 (MAINMOD and SECMOD) and the macro library, MACRO.LIB, which contains over 30 "miniprograms" or *macros.* The back side contains assembled *object programs.*

Having the source allows you to change a program if you want to do so. Having the object allows you to link and run the programs without assembling them.

Programs You Can Execute Directly

Some programs in the book are intended to be used by themselves, rather than with other programs. For these programs, listed in Table E-1, the disk contains a *run file* in addition to the source and object files. Run files, which have the extension EXE, are on the side of the disk with the object programs.

With the DOS prompt (A> or C>) on the screen, you can run any of these programs by simply entering its name. For instance, to play "Turkey In the Straw," enter **turkey** (or **b:turkey**, if the data disk is in drive B).

The disk also contains the source file for EX_PROG, the example program in Chapter 2, but not its object and run files. *You* create those files using the procedure we describe.

Files You Must Link

The remaining disk files are general-purpose procedures you can use in your programs. However, *you must first create a calling program* and assemble it, then link the two object (OBJ) files to form a run (EXE) file. Table E-2 summarizes the information you need to use these files.

Table E-1. Programs that have run (EXE) files.

File Name	*Example in Book**	*Description*
DIAGLINE	7-1	Display a diagonal line
HEX2DEC	--	Convert a hex number to a decimal
KEYMUSIC	8-5	Play music on the keyboard
LOCK	6-5	Write-protect a file
MARY	8-3	Play "Mary Had a Little Lamb"
MOVEBIRD	(7-1)	Move a "bird" across the screen
MOVEFACE	7-2	Move a "smile face" across the screen
NEWDIAG	9-1	Display a diagonal line using macros
RANDFACE	(7-2)	Move a "smile face" across the screen randomly
TURKEY	8-4	Play "Turkey in the Straw"
UNLOCK	6-6	Remove write protection from a file

*Numbers in parentheses indicate answers to Study Exercises. For example, (7-1) means answer #1 for Chapter 7.

Incidentally, some procedures in the book call other procedures. For your convenience, programs on the disk *include* any procedures they call. For example, the MULS32 procedure in Example 4-2 calls MULU32 in Example 4-1. The object file that holds the MULS32 procedure (MULS32.OBJ) includes the instructions for MULU32. This means you never need to link more than two object files: the calling program and the disk program you want to use.

Creating a Calling Program

A calling program is a procedure that executes one or more other procedures, then returns control to the system program you used to execute it (DOS or DEBUG). Every calling program must contain a code segment, a stack segment, and an EXTRN pseudo-op.

The code segment holds the CALL instruction (e.g., CALL MULU32), plus a RET instruction that returns control to DOS. Further, it often has instructions that initialize registers and memory locations the called program expects to receive.

The stack segment reserves memory space for return addresses.

The EXTRN pseudo-op tells the assembler that the procedure being CALLed in the code segment is in another object file on disk. For example, EXTRN MULU32:FAR tells the assembler that the procedure MULU32 is in an object file you will link to this file; the FAR suffix tells it that MULU32 is in a different code segment.

If the called procedure requires you to supply data in a data segment and/or extra segment, your calling program must also contain those segments.

Once you have entered your calling program into the computer and saved it on disk, you assemble it, then link the two object modules to create a run

Table 2. Files you must link.

Link This Object Module	*Call Procedure With This Name*	*Example in Book**	*Description*
ADD_2_OL	ADD_TO_OL	5-7	Add element to ordered list
ADD2UL	ADD_TO_UL	(5-1)	Create an unordered list
ADD_2_UL	ADD_TO_UL	5-1	Add element to unordered list
ASC_BIN	ASCII_BIN	6-7	Convert string to binary number
AVERAGE	AVERAGE	4-3	Average unsigned words
B_SEARCH	B_SEARCH	5-6	Search ordered list
BIN_ASC	BIN_ASCII	6-8	Convert binary number to string
BUBBLE	BUBBLE	5-5	Bubble-sort an unordered list
CLR_S	CLEAR_SCREEN	7-3	Clear lines on the screen
CONV_HEX	CONV_HEX	5-11	Hex to ASCII, BCD, and EBCDIC
COSINE	FIND_COS	5-10	Calculate cosine of an angle
D_SHAPE	DSPLY_SHAPE	7-4	Construct shape from shape table
DEL_OL	DEL_OL	5-8	Delete element from ordered list
DEL_UL	DEL_UL	5-2	Delete element from unordered list
DELAY	DELAY	6-3	Delay for a specified interval
DIVUO	DIVUO	4-4	Divide, account for overflow
M_SHAPE	MOVE_SHAPE	7-5	Move a shape across the screen
MINMAX	MINMAX	5-3	Find min, max in unordered list
MULS32	MULS32	4-2	Multiply 32-bit signed numbers
MULU32	MULU32	4-1	Multiply 32-bit unsigned numbers
PHONE_#	PHONE_NOS	5-13	Sort a telephone list
PLAY	PLAY	8-2	Play a tune
RAND_51	RAND_51	6-1	Random number between 0 and 51
READKEYS	READ_KEYS	6-2	Read a string from the keyboard
REPLACE	REPLACE	(5-2)	Replace element in ordered list
SHOW_ERR	SHOW_ERR	6-4	Display error messages for DOS 2 function calls
SINE	FIND_SINE	5-9	Calculate sine of an angle
SOUND	SOUND	8-1	Generate a sound
SQRT32	SQRT32	4-5	Compute square root by Newton
SR32	SR32	(4-1)	Compute square root by subtracting
TIMEKEYS	TIME_KEYS	(6-1)	Find time between keystrokes

*Numbers in parentheses indicate answers to Study Exercises. For example, (5-1) indicates answer #1 for Chapter 5.

module. For example, Figure E-1 lists a program that calls MULU32, the 32-bit by 32-bit unsigned multiply procedure from Example 4-1. You assemble this program (MULU32C.ASM) to create MULU32C.OBJ, then use the command

```
B>a:link mulu32c+mulu32;
```

to create the run module MULU32C.EXE, which you run under DEBUG.

The program in Figure E-1 is general-purpose; it doesn't tell MULU32 which numbers to multiply. After starting DEBUG, you must use R (Register) commands to load these numbers into CX and BX (multiplier) and into DX and AX (multiplicand).

Since MULU32C doesn't give visual results, the best way to run it is to execute down to the RET instruction, then examine the registers. If you assemble MULU32C and display (*type*) the list file (MULU32C.LST), you discover that the RET instruction has an offset of 000A. Hence, you run the program with the DEBUG command *ga* (for "Go to A").

```
TITLE   MULU32C - MULU32 Caller
        PAGE    ,132
        EXTRN   MULU32:FAR  ;MULU32 is an external procedure
;
;  Set up the stack segment.
;
STACK   SEGMENT PARA STACK 'STACK'
        DB      64 DUP('STACK   ')
STACK   ENDS
;
;  Here is the code for the calling program.
;
CSEG    SEGMENT PARA PUBLIC 'CODE'
CALLER  PROC    FAR
        ASSUME  CS:CSEG,SS:STACK
;
;  Set up the stack so this program returns to DEBUG.
;
        PUSH    DS          ;Put return seg. addr. on stack
        MOV     DI,0        ;Clear a register
        PUSH    DI          ;Put zero return addr. on stack
;
;  Call MULU32
;
```

```
        CALL    MULU32
        RET                     ;Return to DEBUG
CALLER  ENDP
CSEG    ENDS
        END     CALLER
```

Figure E-1. Program that calls MULU32.

Index

$ (location counter) operator, 47
* (multiply) operator, 44
& operator, 296-297
+ (add) operator, 44
− (subtract) operator, 44
/ (divide) operator, 44
; (comment designator), 28
;; (macro comment) operator, 297;

8087 math coprocessor, 353-369
  data types, 355
  instruction set, 356-358
  internal registers, 354-355
  programming the, 358-359
  stack, 354
8088 microprocessor, 10-20
  compared to 8086, 10
  evolution of, 10-11
  internal registers, 15-20
  overview of, 11-15
8253 system timer, 207
8255 PPI chip, 275
8259 interrupt controller, 207

Accumulator register (AX), 15
Adapters, 257-258
  Color/Graphics, 257-258
  Monochrome, 257
Adding binary numbers, 4
Addition instructions, 102-105
Address, physical, 11
Address bus, 15
Address transfer instructions, 97-99
Addressing modes, 79-88
  base indexed, 86-88
  base relative, 85
  direct, 83-84
  direct indexed, 85-86
  immediate, 81-82
  register, 81
  register indirect, 84-85
  table of, 81
Addressing, memory, 11-12
Alphanumeric characters, 257
Alt-Ctrl-Del key combination, 243
AND operator, 45
Animation, 261-266
Arithmetic data formats, 101-102
  binary numbers, 101
  decimal numbers, 101-102
Arithmetic instructions, 100-114
Arithmetic operators, 44
Arranging numbers in increasing order, 131-132
ASCII
  converting hex to, 194-195
  Extended, 236
  display characters, 258-260
  text, 23
  to binary code conversions, 247-255
Assembler, 23
Assembler disk, creating, 50-51
Assembly language instructions, format of, 26-29
Assembly procedure, 55
Asynchronous communications functions, DOS, 220
Attribute operators, 48-50
Attributes
  character, 258-260
  file, 231
Auxiliary Carry Flag (AF), 19
Averaging words in memory, 161-162
AX register, 15

Base indexed addressing mode, 86-88
Base register (BX), 15

Base relative addressing mode, 85
BASIC, Cassette, 215
Batch file for SALUT, 350-351
Beep procedure, BIOS, 276
Binary-coded decimal (BCD) numbers
  adding, 104-105
  dividing, 113-114
  multiplying, 112
  packed, 101
  subtracting, 108-109
  unpacked, 101
Binary digits (bits), 2-6
  adding, 4
  weights of, 3
Binary numbering system, 1-6
Binary constants, 25
Binary search for ordered lists, 181-185
Binary suffix (B), 25
BIOS interrupts, 202-218
  8088 interrupt vectors, 205-207
  8259 interrupt vectors, 207-208
  BIOS entry points, 208-217
  data table pointers, 218
  user-supplied routines, 217-218
Bit manipulation instructions, 115-121
Bit positions, hexadecimal values for, 116
Bits, 2
Blinking characters, 258-259
Brackets, 26
Breakpoint interrupt (Type 3), 206
Bubble sort, 175-181
Bus Interface Unit (BIU), 17
Buses, 15
BX register, 15
Byte (8-bit value), 3

Calling procedures, 123
Carry Flag (CF), 18
Cassette BASIC interrupt (Type 18), 215
Cassette I/O interrupt (Type 15), 214
Character(s)
  alphanumeric, 257
  ASCII display, 258-260
  attributes, 258-260
  blinking, 258-259
  codes, keyboard, 240-244
  constants, 26
  in labels, 27
Chip, 9
Clear the screen, 268
Clock cycle, 13
Clock speed, 13
Code conversions, ASCII/binary, 247-255
Code segment, 15
Code segment register (CS), 17
Colons (:) in labels, 27
Color/Graphics Adapter, 257-258
COM file, 70
Comment field, 28-29
Communications functions, DOS, 220
Communications I/O interrupt (Type 14), 214
Compare instructions, 110
  using conditional transfers, 131
Compare-string instructions, 140-141
Conditional pseudo-ops, 71-74, 294-295
Conditional transfer instructions, 128-132
  compares used with, 131
Constants in source statements, 25-26, 354-355, 359
Control transfer instructions, 121-133
Converting
  ASCII string to binary, 247-254
  binary number to ASCII, 254-255
  decimal to binary, 2
  decimal to hex, 371
  hex to ASCII, 194-195
  hex to BCD, 194-195
  hex to decimal, 7, 371
  hex to EBCDIC, 194-195
Coprocessor, 80, 87, 353
Cosine of an angle, 194
Cross-reference listing, 64
Cross-reference table, macro library, 310
Crystal, 13
CS (code segment) register, 17
CS: (segment override operator), 49
Ctrl-Break interrupt, 217
Ctrl-Num Lock Key combination, 243
CX register, 16

Cycles per second, 13

Data bus, 15
Data disk, 51
Data pseudo-ops, 30-39, 359
  table of, 30-31
Data registers, 15-16
Data segment, 15
Data segment register (DS), 17
Data structures, definition of, 169
Data transfer instructions, 91-100
Data types, 8087 math coprocessor, 355
Date and time functions, DOS, 226-230
DEBUG
  commands, 60-61
  running programs under, 60-63
Decimal suffix (D), 25
Decimal constants, 26
Decimal numbers, See *Binary-coded decimal (BCD)*
Decision sequence, three-way, 132
Delay
  generating a, 227-230
  macro, 303
Digit(s)
  binary, 2
  hexadecimal, 7
Direct addressing mode, 83-84
Direct indexed addressing mode, 85-86
Direction Flag (DF), 19, 135
  use in string instructions, 135
Directory functions, DOS, 230-231
Directory of an object library, 329
Disk directory functions, DOS, 230-231
Disk files, write-protecting, 234-236
Disk I/O interrupt (Type 13), 214
Display
  characters, ASCII, 258-260
  functions, DOS, 224-225
  modes, 257-258
Distance attributes (NEAR and FAR), 36-38
Divide by zero interrupt, 205
Division
  high-precision, 161-165
  instructions, 112-114
  without overflow, 162-165
DO structure, 338-343
DOS interrupts, 218-236
DOS 2 error message program, 232-234
Doubleword (32-bit value), 32
DS (data segment) register, 17
DS: (segment override operator), 49
DX register, 16

EBCDIC, converting hex to, 194-195
Editor, 22-23
EDLIN line editor, 53-55
Effective Address (EA), 83
Equipment check interrupt (Type 11), 213
Error message program, DOS 2, 232-234
ES (extra segment) register, 17
ES: (segment override operator), 49
Execution time, calculating, 226-227
Execution Unit (EU), 17
Expression, 31
Extended ASCII, 238
Extended codes, keyboard, 242-243
Extended file management functions, DOS, 231-232
Extra segment, 15

FAR procedures, 37
Fields, instruction, 26-29
File attributes, 231
File handle, 231-232
File management functions, DOS, 226
Files, write-protecting, 234-236
Fixed disk interrupt (Type D), 208
Flag operations, 148
Flag transfer instructions, 99-100
Flags register, 18-20
Floating-point format, 354-355
Floppy disk interrupt (Type E), 208
Flowchart, 22

Handle, file, 231-232

Hertz (cycles per second), 13
HEX2DEC.EXE, 7
Hexadecimal constants, 26
Hexadecimal digits, 7
  weights of, 7
Hexadecimal numbers, 6-8
Hexadecimal suffix (H), 25
Hexadecimal values for bit positions, 116
HIGH operator, 49

IF structure, 334-338
Immediate addressing mode, 81-82
Indenting programs using SALUT, 352
Index registers, 17
Indirect calls to procedures, 125-126
Input/output instructions, 97
Input/output ports, 13
Instruction pointer, 17-18
Instruction set, 8087 math coprocessor, 356-358
Instruction types, 88-91
Instructions, list of, 89-91
  See also *Quick Index*
Interrupt Enable Flag (IF), 19
Interrupt
  instructions, 144-147
  non-maskable, 206
  vector functions, DOS, 230
  vector(s), defined, 14
Interrupts, 14-15
Iteration control (loop) instructions, 132-133

Jump tables, 196-197

K (1024), 4
Key combinations, special, 243-244
Keyboard
  break interrupt (Type B), 217
  character codes, 240-244
  extended codes, 242-243
  operation of, 236-237
  reading strings from the, 225
  functions, DOS, 219, 224, 246
I/O interrupt (Type 16), 214, 244-246
interrupts, 207, 244-246
operations, 237-246
scan codes, 240-244

Label field, 26-27
Labels
  characters in, 27
  using colons with, 27
Labels on END pseudo-ops, 39
LENGTH operator, 48
Library, macro, 299-326
Link procedure, 56, 60
Linker (LINK), 23-24
Linking multiple object modules, 60
Linking object libraries, 329
Listing pseudo-ops, 39-40, 296
Listing source programs, 55-56
Lists, See *Unordered lists* and *Ordered lists*
Load-string instruction, 143
Logic flow structures, 332
Logical instructions, 115-118
Logical operators, 44-46
Look-up tables, 189-197
Looping, 132-133
LOW operator, 49

Macro Assembler, 21-22
Macro library, 299-326
  creating a, 300
  cross-reference table, 310
  reading into a program, 300-301
Macro operators, 296-297
Macro pseudo-ops, 290-296
  table of, 291-292
Macros
  advantages of, 287-288
  compared to procedures, 287-288
  contents of, 289-290
  defining in source programs, 297-299
  purging, 301
MAINMOD.ASM (program model), 66
Map listing, 64-65

Mary Had a Little Lamb, 281-282
Maximum and minimum value in list, 172-175
Memory, format of numbers in, 102
Memory addressing, 11-12
Memory map, 203
Memory size interrupt (Type 12), 213
Mnemonic field, 28
MOD operator, 44
Models, program, 65-68
Modes, display, 257-258
Monochrome Adapter, 257
Move-string instructions, 137-138
Moving blocks of memory, 137-138
Multiplication
  high-precision, 155-161
  instructions, 111-112
  signed 32-bit × 32-bit, 159-161
  unsigned 32-bit × 32-bit, 156-159
Music through the speaker, 278-286

Nanosecond, 13
NEAR procedures, 36-37
Negative numbers, 5-6, 26
  entering, 26
Nesting conditional clauses, 73-74
Nesting procedures, 126-127
Non-maskable interrupt (Type 2), 206
NOT operator, 46
Numbers, signed, 4-6
Numbers in memory, format of, 102

Object library
  building an, 327-328
  directory of an, 329
  linking an, 329
  operating on an, 328-329
Object program, 21
Offset, 12
OFFSET operator, 47
Operand field, 28
Operands, 28
Operators, 41-50, 296-297
  macro, 296-297
OR operator, 45
Ordered lists, 181-188
  adding elements to, 185-187
  deleting elements from, 187-188
  searching, 181-185
Overflow, dealing with, 162-165
Overflow Flag (OF), 19
Overflow interrupt (Type 4), 206
Override operators, 49
Overriding segment assignments, 139

Packed BCD numbers, 101
Parity Flag (PF), 19
Pels (picture elements), 257
Physical address, 11
Pipeline, 18
Pointer registers, 17
Ports, input/output, 13
Power-on reset interrupt (Type 19), 215
Prefixes
  repeat, 136
  segment override, 139
Print screen interrupt (Type 5), 206-207
Printer I/O interrupt (Type 17), 214
Procedures, 36, 122-123, 287-288
  compared with macros, 287-288
  indirect calls to, 125-126
  nesting, 126-127
Processor control instructions, 147-149
Program execution time, calculating, 226-227
Program models, 65-68
Pseudo operations (pseudo-ops), See *Quick Index*
PTR operator, 48
Purging macros, 301

Random number macro, 303
Random numbers, generating, 216-217
Reading strings from the keyboard, 225
Register addressing mode, 81
Register indirect addressing mode, 84-85
Registers
  8088, 15-20
  8087 math coprocessor, 354-355
  data, 15-16

flags, 18-20
index, 17
instruction pointer, 17-18
pointer, 17
segment, 16-17
Relational operators, 46-47
Relocatable programs, 23
Repeat prefixes, 136
Repeat pseudo-ops, 293-294
Reset, power-on, 215
Rotate instructions, 121
Run file, creating, 56, 60
Running programs, 60-63

SALUT program
batch file for, 350-351
described, 332
indenting programs using, 352
using, 350-351
Scan codes, keyboard, 240-244
Scan-string instructions, 142-143
Screen, clearing the, 268
SEARCH structure, 343-348
Searching ordered lists, 181-185
SECMOD.ASM (program model), 68
SEG operator, 47
Segment, 15
Segment assignments, overriding, 139
Segment number, 12
Segment override
operators, 49
prefixes, 139
Segment registers, 16-17
Segments for string instructions, 134
Shape tables, 266-272
Shift instructions, 119-121
Shift PrtSc key combination, 243
SHL operator, 44
SHORT operator, 49
SHR operator, 44
Sign Flag (SF), 19
Sign-extending immediate values, 82
Sign-extension instructions, 114
Signed numbers, 4-5
Sine of an angle, 189-193
Single-step interrupt (Type 1), 205-206
Single-stepping through programs, 62-63
SIZE operator, 48
Small Assembler, 21-22
Sort, bubble, 175-181
Sorting a telephone list, 198-200
SOUND procedure, 277-278
Source program, 21
Source statement, 23, 25-26
constants in, 25-26
Speaker
music through the, 278-286
operation of, 275
programming the, 275-278
Special key combinations, 243-244
Square root, 165-167
SS (stack segment) register, 17
SS: (segment override operator), 49
Stack
8087 math coprocessor, 354
effect of PUSH and POP on, 93-96
segment, 15
Stand-alone comments, 29
Statement, source, 23
Store-string instruction, 144
String constants, 26
String instructions, 134-144
segment assumptions for, 134
Structured programming
DO structure, 338-343
IF structure, 334-338
SEARCH structure, 343-348
Structured programs, preparing, 349-352
Subtraction
in 8088, 106-107
instructions, 106-110
'Symbol is Multi-Defined' error message, 290
System memory, 201-202
System timer interrupt (Type 8), 207

Table look-up instruction (XLAT), 96-97
Tables
jump, 194-195
shape, 266-272
Telephone list, sorting, 198-200

Text files, 197-200
THIS operator, 49
Time and date functions, DOS, 226-230
Time of day interrupt (Type 13), 215-216
Timer tick, 207, 217
  interrupt (Type 1C), 217
Top-down design, 24-25, 349-350
Tracing through programs, 62-63
Trap Flag (TF), 19
Turkey in the Straw, 283
Two's-complement, 5-6, 26
  how to calculate, 106
  instruction (NEC), 109-110
Type 0 (Divide by zero) interrupt, 205
Type 1 (Single-step) interrupt, 205-206
Type 2 (Non-maskable) interrupt, 206
Type 3 (Breakpoint) interrupt, 206
Type 4 (Overflow) interrupt, 206
Type 5 (Print screen) interrupt, 206-207
Type 8 (8253 System Timer) interrupt, 207
Type 9 (Keyboard) interrupt, 207
Type D (Fixed disk) interrupt, 208
Type E (Floppy disk) interrupt, 208
Type 10 (Video I/O) interrupt, 208-212
Type 11 (Equipment check) interrupt, 213
Type 12 (Memory size) interrupt, 213
Type 13 (Disk I/O) interrupt, 214
Type 14 (Communications I/O) interrupt, 214
Type 15 (Cassette I/O) interrupt, 214
Type 16 (Keyboard I/O) interrupt, 214, 244-246
Type 17 (Printer I/O) interrupt, 214-215
Type 18 (Cassette BASIC) interrupt, 214-215
Type 19 (Power-on reset) interrupt, 215-217
Type 1B (Keyboard break) interrupt, 217
Type 1C (Timer tick) interrupt, 217
Type 21 function calls, 219-232
  error codes for, 223
  table of, 219-223
TYPE operator, 47

Unconditional transfer instructions, 122-128
Unordered lists, 169-181
  adding elements to, 170-171
  deleting elements from, 171-172
  maximum and minimum value in, 172-175
  sorting, 175-181
Unpacked BCD numbers, 101

Value-returning operators, 47-48
Vector, 14, 34
Video I/O interrupt (Type 10), 208-212

Weights
  of binary digits, 3
  of hexadecimal digits, 7
Word (16-bit value), 32
Write-protecting files, 234-236

XOR operator, 45

Zero Flag (ZF), 19

Quick Index

Instructions

AAA, 104
AAD, 113
AAM, 112
AAS, 108
ADC, 102
ADD, 102
AND, 116
CALL, 123
CBW, 114
CLC, 148
CLD, 135, 148
CLI, 148
CMP, 110, 131
CMPS, 140
CMPSB, 142
CMPSW, 142
CWD, 114
DAA, 104
DAS, 108
DEC, 109
DIV, 112
ESC, 149
HLT, 148
IDIV, 112
IMUL, 111
IN, 97
INC, 105
INT, 145
INTO, 146
IRET, 147
JMP, 127
Jx, 129
LAHF, 99
LDS, 98
LEA, 98
LES, 99
LOCK, 149
LODS, 143
LODSB, 143
LODSW, 143
LOOP, 133
LOOPE, 133
LOOPNE, 133
LOOPNZ, 133
LOOPZ, 133,
MOV, 92
MOVS, 137
MOVSB, 138
MOVSW, 138
MUL, 111
NEG, 109
NOP, 149
NOT, 118
OR, 116
OUT, 97
POP, 93
POPF, 99
PUSH, 93
PUSHF, 99
RCL, 121
RCR, 121
RET, 123
ROL, 121
ROR, 121
SAHF, 99
SAL, 119
SAR, 119
SBB, 107
SCAS, 142
SCASB, 143
SCASW, 143
SHL, 119
SHR, 119
STC, 148
STD, 135, 148
STI, 148
STOS, 144
SUB, 107
TEST, 118
WAIT, 149
XCHG, 96
XLAT, 96
XOR, 116

Pseudo-ops

%OUT, 75
.CREF, 75
.LALL, 296
.LFCOND, 75
.LIST, 75
.SALL, 296
.SFCOND, 75
.XALL, 296
.XCREF, 75
.XLIST, 75
=, 32
ASSUME, 35
DB, 33
DD, 34
DW, 34
END, 39
ENDP, 36
ENDS, 35
EQU, 32
EVEN, 70
EXITM, 295
EXTRN, 38
GROUP, 69
IF, 72
IF1, 72, 292
IFB, 294
IFDEF, 72
IFE, 72
IFIDF, 72
IFIDN, 72
IFNB, 294
IFNDEF, 72
INCLUDE, 39, 300
IRP, 293
IRPC, 293
LABEL, 70
LOCAL, 290
MACRO, 289
ORG, 70
PAGE, 39
PROC, 36
PUBLIC, 38
PURGE, 301
REPT, 293
SEGMENT, 35
SHORT, 127
SUBTTL, 40
TITLE, 40

NOW INCLUDES IBM MACRO ASSEMBLER VERSION 2.00!

IBM PC & XT Assembly Language: A Guide For Programmers ENHANCED AND ENLARGED

Leo J. Scanlon

". . .best introductory text for assembly language programmers."
PC TECH JOURNAL

"An excellent book. . .I will recommend it to my readers."
DR. DOBB'S JOURNAL

Take direct control of your IBM PC through the power of assembly language. The new edition of this classic bestseller will enhance and reveal the capabilities of your machine in detailed, understandable steps. You will learn to develop assembly language programs with explanations that are sensitive to your skills and level of experience. Beginners will appreciate the graduated pace and introductory background information. For the more advanced programmers, the guide provides sufficient technical detail for high-level programming . . .Includes coverage of IBM Macro Assembler 2.00!

IBM PC & XT Assembly Language contains these special features:

- Step-by-step instructions for using the IBM Macro Assembler (Versions 1.00 and 2.00), the EDLIN line editor, and the DEBUG debugging program.
- An inside look at BIOS, the PC's "chief administrator," which can save you hours of programming time.
- New chapters on MACROS (including a library of over 30 macros), object libraries, and structured programming.
- Handy techniques for multi-precision arithmetic, sorting, and producing sound and graphics.
- Valuable applications programs, summary tables and illustrations.

CONTENTS

A Crash Course in Computer Numbering Systems • Introduction to Assembly Language Programming • Using an Assembler • The 8088 Instruction Set • High Precision Mathematics • Operating on Data Structures • Using the System Resources • Graphics Made Easy • Let There Be Sound! • Macros • Object Libraries • Structured Programming • 8087 Math Coprocessor • Answers to Study Exercises • Appendices • Disk User's Guide • Index

ISBN 0-89303-575-0